Enhanced Server Fault-Tolerance Techniques for Improved User Experience

by

Manish Marwah

B.Tech., Indian Institute of Technology, Delhi, 1993

M.S., University of Colorado, Boulder, 1996

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2008

This thesis entitled:
Enhanced Server Fault-Tolerance Techniques for Improved User Experience
written by Manish Marwah
has been approved for the Department of Computer Science

____
Shivakant Mishra

____
Prof. Christof Fetzer

Date ____________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Marwah, Manish (Ph.D., Computer Science)

Enhanced Server Fault-Tolerance Techniques for Improved User Experience

Thesis directed by Prof. Shivakant Mishra

User applications, such as email, calendar, maps, are migrating from local desktop machines to data centers due to the many advantages offered by such a computing paradigm. 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 HW, inexpensive commodity machines - although having low availability numbers - are the most cost effective choices for a data center. However, increased server failures cause service outages and degrade user experience which in turn results in lost revenue for businesses. Also, emerging web applications put additional demands on server fault-tolerance. For example, if a user is browsing a map service like Google, Yahoo or MSN maps, a server failure leading to an outage of more than a few seconds is detectable by a user and hence degrades user experience.

In this thesis, I propose three novel techniques aimed at improving server fault-tolerance: (1) ST-TCP, which is an extension of TCP to tolerate server failures. This is done by using an active-backup which replicates the state of a primary and seamlessly takes over a TCP connection on primary server failure; (2) CRAFT, where the TCP splicing mechanism is enhanced to make it both fault-tolerant and more scalable; this then forms the basis of a scalable and fault-tolerant web server architecture that specifically addresses server fault-tolerance issues for highly interactive or real time applications; and, (3) Call-preserving failover, which is an efficient and scalable fault-tolerance mechanism for migrating IP telephony calls to an alternate call controller.

Dedication

To my Parents

Acknowledgements

I would like to thank my advisor, Prof. Shivakant Mishra, who supported my research and provided invaluable guidance through the years. He gave me freedom to define my thesis research problem and its scope, while being always accessible to discuss ideas.

Prof. Christof Fetzer has been instrumental during the course of this research. Although many time zones away, he would always be very prompt in providing insightful feedback that advanced my research in more ways than he is probably aware of. Thank you!

I would also like to thank the other members of my thesis committee: Prof. Dirk Grunwald, Prof. Rick Han, Prof. John Black and Dr. P. Krishnan. I especially want to thank Dirk for making available machines that were used for the experiments conducted in the latter part of the thesis.

Prof. Roop Mahajan, my Master's thesis advisor, has been a mentor and a constant source of encouragement.

My stay at CU was made memorable by the good times shared with many friends, particularly Kishore Yellampalle and Anand Narayan.

Finally, I want to thank my family for their constant support and encouragement during the course of my research.

Contents

Chapter

1 Introduction 1

1.1 Motivations for Client-Transparent Fault-Tolerance 4

1.2 Main challenges 6

1.2.1 Transport layer challenges 6

1.2.2 Application layer challenges 7

1.2.3 Some Examples 8

1.2.4 Desirable System Features 11

1.3 Contributions 13

1.3.1 ST-TCP 13

1.3.2 CRAFT 15

1.3.3 Migration of IP Telephony Calls 17

1.4 Organization 17

2 Related Work 18

2.1 Primary-backup Systems 18

2.2 Transport Layer Techniques 19

2.2.1 FT-TCP 19

2.2.2 HydraNet 21

2.2.3 Orgiyan and Fetzer 21

2.2.4 M-TCP 21
2.2.5 TCP Migrate 22
2.2.6 SCTP 23
2.2.7 Swift 23
2.2.8 TCP Splice 23
2.2.9 Backdoors 24
2.3 Group Communication 24

3 Server fault-Tolerant TCP 25
3.1 Requirements 25
3.2 Architecture Overview 27
3.2.1 Ethernet Tapping 28
3.2.2 System Architecture 30
3.3 Protocol Details 32
3.3.1 Failure-free Period 33
3.3.2 Synchronizing The Backup Server 37
3.3.3 Failure Detection 38
3.4 Implementation 39
3.5 Performance 40
3.5.1 Performance Overhead of ST-TCP 41
3.5.2 Failover Time in ST-TCP 42
3.6 Enhancements to ST-TCP 44
3.6.1 Heartbeat Mechanism 46
3.6.2 Application Crash Failure 47
3.7 Failure Scenarios and Recovery Actions 48
3.7.1 HW/OS Crash Failures 50
3.7.2 Application Crash Failures 50

3.7.3 Application Crash Failure withoutCleanup 50
3.7.4 Application Crash Failure withCleanup 51
3.7.5 Local Network Failures 55
3.8 Summary 57

4 Fault-Tolerant and Scalable TCP splice 59
4.1 Introduction 59
4.2 Background 61
4.2.1 TCP Splice 61
4.2.2 Related Work 62
4.3 Architecture overview 63
4.4 Flexible Server Configurations 67
4.5 Load Balancing 69
4.5.1 Load Balancing Using Channel Bonding 70
4.5.2 L3 Load Balancer 71
4.6 Proxy Architecture 73
4.7 Implementation 75
4.7.1 Load Balancer 75
4.7.2 Proxy 76
4.7.3 Backend Server 79
4.8 Experimental results 79
4.8.1 Experimental Setup 79
4.8.2 Proxy scalability 81
4.8.3 Proxy Failovers 84
4.8.4 TCP Split Splice 86
4.9 Summary 87

5 Fault-Tolerant Web Server Architecture 88

5.1 Introduction 88

5.2 Background 91

5.2.1 TCP splice 91

5.2.2 Enhancements to TCP splice 92

5.2.3 Related work 93

5.3 System architecture: An Overview 94

5.3.1 Logging, Transactionalization and Tagging 97

5.3.2 Synchronization and Re-splicing 99

5.3.3 Application Support 100

5.3.4 Non-determinism 101

5.3.5 Adaptive Failure Detection 102

5.4 Detailed Design 103

5.4.1 Normal Operation 103

5.4.2 Terminology 104

5.4.3 Failure Recovery 105

5.5 Implementation 107

5.6 Experiments and Performance Evaluation 109

5.6.1 Experimental Setup 110

5.6.2 Experiments 111

5.6.3 Results and Discussion 113

5.7 Summary 116

6 Systems Architectures for Transactional Network Interface 118

6.1 Introduction 118

6.2 Example Application of a Transactional Network Interface 121

6.3 System Architecture 124

6.3.1 Transport layer architecture 125

6.3.2 Two Connection Application layer architecture 132

6.3.3 Single machine application layer architecture 133

6.4 Implementation Details 135

6.4.1 Transport layer architecture 135

6.4.2 Application layer architectures 139

6.5 Experiments and performance evaluation 139

6.5.1 Experimental Setup 140

6.5.2 Experiments 141

6.5.3 Discussion of results: LAN Scenario 142

6.5.4 Discussion of Results: WAN Scenarios 145

6.6 Summary 147

7 Efficient, Scalable Migration of IP Telephony Calls for Enhanced Fault-tolerance 149

7.1 Introduction 149

7.2 Architecture overview of a IP telephony system 151

7.3 Requirements/Goals 153

7.4 Call Migration 154

7.4.1 Saving State on Clients 156

7.4.2 Reconstructing Calls 156

7.4.3 Comparison to alternatives 157

7.4.4 Advantages 158

7.5 Merging call components during reconstruction 159

7.5.1 Advantages of merging a call 160

7.5.2 Call components merging algorithm 160

7.6 Summary 162

8 Conclusions and Future Work 165

Bibliography 168

Tables

Table

1.1 Estimated cost of service outage for some businesses (taken from [57] and [79]). 2

3.1 Comparison of standard TCP with ST-TCP during failure free period. 42

3.2 ST-TCP failover time for the three applications. 45

3.3 Single Failure Scenarios 49

4.1 Specifications of the machines used in the experiments. 80

4.2 Experimental results for 100 concurrent client PUT requests of a file of size 10 MB 83

4.3 Experimental results for 100 concurrent client PUT requests of a file of size 50 MB 84

4.4 Experimental results for 100 concurrent PUTs of 10 MB with a proxy failure during the transfer. 85

4.5 Experimental results for 100 concurrent GET of 10 MB with **split splice** at the server. 87

4.6 Experimental results for 100 concurrent GET of 50 MB with **split splice** at the server. 87

5.1 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. 111

5.2 Time taken for performing one run of Action 1: Connecting to the login screen. 114

5.3 Time taken for performing one run of Action 2: Logging in; drafting and sending an email; and, logging out. 114

6.1 Average transaction rate for the three architectures in the LAN scenario. Also listed are the average, minimum, and maximum times taken for a transaction. 142

6.2 Average transaction rate for the three architectures in the WAN1–MIT scenario. Also listed are the average, minimum, and maximum times taken for a transaction. 145

6.3 Average transaction rate for the three architectures in the WAN1–SG scenario. Also listed are the average, minimum, and maximum times taken for a transaction. 147

8.1 Comparison between the three server fault-tolerance mechanisms. 166

Figures

Figure

1.1 (a) Primary server with a backup. The application on the primary synchronizes session state information with the backup. However, without support from the client, just synchronizing session state with the backup is not sufficient to preserve the session. (b) After failure of the primary server, backup takes over; client establishes a new TCP connection with the backup. 3
1.2 A user downloading a file using a browser. 9
1.3 A Samba client connecting to a Samba server remotely over a VPN tunnel. . . 10
2.1 Wrappers in FT-TCP. 19
3.1 Tapping TCP byte stream. 27
3.2 Ethernet switch tapping architecture. 29
3.3 System Architecture without a single point of failure. 30
3.4 Receive buffer of TCP server: (a) Standard TCP, (b) ST-TCP. 35
3.5 Performance of (a) Echo (upper line: with failure; lower line: without failure) and (b) Interactive (upper line: with failure; lower line: without failure). . . . 43
3.6 Bulk Transfer Application run with ST-TCP. Graph shows the total time taken with a failover and without failure (lower curve with same line type) during a run for data transfers of sizes 1 MB, 5MB, 20MB and 100MB. 44
4.1 Functional components of our architecture. 64

4.2 Some examples of possible server configurations. (a) Separate proxy machines; no backend server changes required. The figure shows the paths of packets belonging to a single connection. Packets in both directions need to pass through a proxy, but it could be any proxy. (b) Proxy and backend server on the same machines. A single connection is shown. Here the response packets are always directly sent to the clients. (c) Mixed configuration with separate proxies, and, split splice installed on some backend servers. The figure shows two connections, one on each backend server. 68

4.3 Channel bonding used for ports on an L2 switch. 70

4.4 A layer 3 load balancer. 71

4.5 Sequence of steps involved in handling a client request. 74

4.6 Netfilter allows functions to be registered at five different hooks in the network stack. 76

4.7 Pseudocode of the proxy implementation. 78

4.8 Experimental setup. 80

4.9 The plot shows the increase in the network throughput as more proxies are added. 83

5.1 Components of our Web Server Architecture. 94

5.2 Sequence of steps required in handling a client request. 103

5.3 Experimental setup. 110

6.1 Use of socket read/write within an atomic STM block. 121

6.2 Compiler generated code for the code segment shown in Figure 6.1 . . . 123

6.3 System architecture. 125

6.4 States assigned to a transactional TCP connection at the logger. 129

6.5 Sequence and ack number adjustments are required due to transaction aborts. 131

6.6 Application level architecture with two TCP connections. 132

6.7 Application level architecture with no separate logger. 134

6.8 Hooks in netfilter. 136

6.9 Data structure for storing state information (mainly sequence number related) for each direction of a TCP connection. 138

6.10 Experimental setup. 140

6.11 Transactional rate versus percentage of first-attempt commit transactions for the **LAN scenario**. 143

6.12 Communication and computational costs of a transaction for TLA in the LAN scenario. 144

6.13 Transactional rate versus percentage of first-attempt commit transactions for the **WAN1-MIT scenario**. 146

6.14 Transactional rate versus percentage of first-attempt commit transactions for the **WAN2-SG scenario**. 148

7.1 Functional architecture of an IP telephony system 152

7.2 An IP telephony system. 155

7.3 An inter-gateway connection. 159

7.4 (a) A call between a trunk, T1 and a phone, P1 is created. (b) After failure of the CC, the VGs migrate to the alternate CC. The original call gets reconstructed as two different calls. 163

7.5 A call between P1 and P2 is shown. After failure of the CC, VG1 and VG2 migrate to the alt. CC, which uses the H.248 properties saved at the IGC terminations to merge the two ends of the call. 164

7.6 This shows a three party call. After the failure of the CC, the order in which the VGs migrate to the alt. CC is significant. The alt. CC uses the merge algorithm described here to merge the three parts of the call. 164

Chapter 1

Introduction

As a society, our dependence on the Internet has grown remarkably in the past ten years. From being used only in academic and research settings, now its use is prevalent in everyday life. In fact, the Internet is being viewed today as a public utility similar to electricity, telephony and cable TV services.

For a vast majority of people, the Internet is the major source for communication, shopping, information, education, and news. Its impact on the functioning of businesses and the government is also formidable. For instance, in 2005, for the first time a majority of US income taxes were filed electronically.

Given the ubiquity of computers and Internet connectivity, and, ease of use of Internet browsers, e-commerce has hit prime time. Products and services sold over the Internet are a substantial part of the revenue of many companies. In fact, there are a significant number of companies that solely depend on the Internet for all their revenues.

Since the Internet has become such a focal part of our lives, its reliability assumes great significance. For companies that depend heavily on the Internet for their sales, service outages can be very expensive.

The average cost of downtime for a corporation is estimated to be about $8000 per hour [79], but it is substantially higher for large corporations as indicated in Table 1.1; additionally, outages lead to other adverse economic consequences for companies like a significant drop in their stock value and future sales. For instance, E-Trade and Ebay

took a hit of 22% and 20% in their market capitalization, respectively, after outages at their web sites a few years back [79]. Furthermore, degraded or slow service is also not acceptable as it causes customers to go to a competitor's more responsive web site.

Business	Per Hour Cost of Downtime
Brokerage operations	$6,450,000
Credit card authorization	$2,600,000
Ebay	$225,000
Amazon.com	$180,000
Package shipping services	$150,000
Home shopping channel	$113,000
Airline reservation center	$89,000
Cellular service activation	$41,000
ATM service fees	$14,000

Table 1.1: Estimated cost of service outage for some businesses (taken from [57] and [79]).

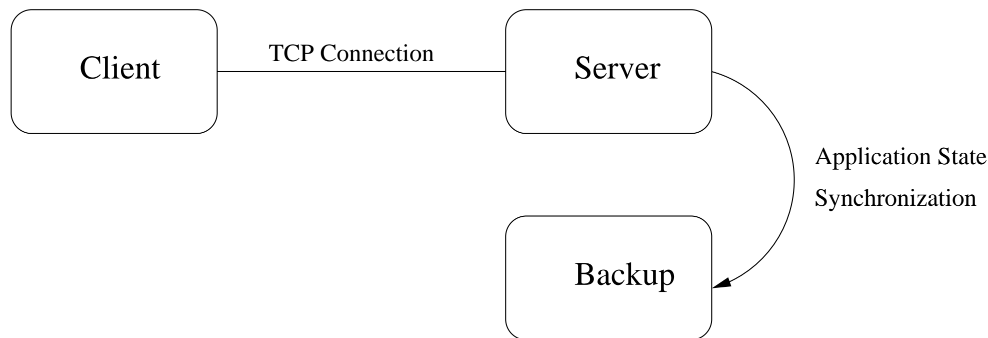
Thus, any improvements made towards eliminating (or limiting) service outages and degradations can have substantial economic benefits.

There are various causes of Internet service outages [54]. In this thesis, we look at one such cause: failure of a server. More specifically, we examine how seamless, client-transparent fault-tolerance can be provided despite server failures in data centers. A key constituent of achieving such fault-tolerance is mechanisms for client-transparently tolerating failures of a TCP connection due to a server failure.

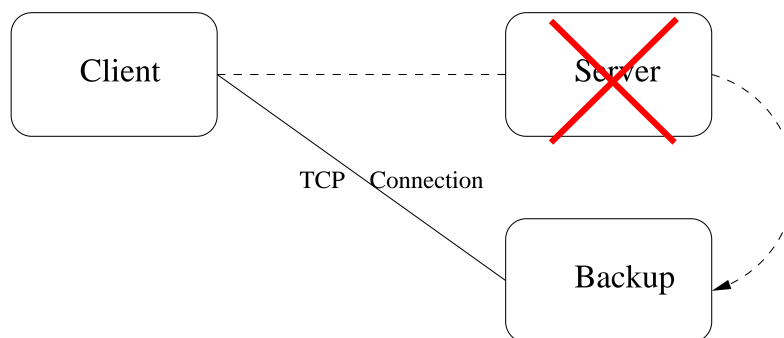
Transmission Control Protocol (TCP) [58] is the most important transport layer protocol used in the Internet infrastructure. It is used in a wide range of popular, distributed applications such as web browsers (http), ftp, telnet, ssh, sendmail and Samba. The main reason for TCP's popularity is its rich set of desirable features, including a reliable, ordered, duplex byte stream, flow control, and congestion control. However, an important feature that TCP does not provide is server fault tolerance.

Most distributed applications on the Internet are based on the client-server paradigm. To enhance the reliability of the system, an additional server (a backup server) may be

deployed to provide service in the event of the failure of the primary server. If the machine on which a server application is running fails, all TCP connections to the server break and all TCP clients get disconnected from the server. Typically, when the system recovers (the failed server reboots and recovers, or a backup server takes over), the client re-establishes the TCP connection and a new session is created.



(a)



(b)

Figure 1.1: (a) Primary server with a backup. The application on the primary synchronizes session state information with the backup. However, without support from the client, just synchronizing session state with the backup is not sufficient to preserve the session. (b) After failure of the primary server, backup takes over; client establishes a new TCP connection with the backup.

It is possible to revive the failed session, however, to do so both the client and the server applications must support a mutually agreed upon fault-tolerance mechanism.

This mechanism would allow the client and the server to re-synchronize state on TCP connection re-establishment. This is necessary to determine the bytes that may have been lost during the failure (e.g. already acknowledged client bytes in the server TCP receive buffers, not yet read by an application, are lost).

With the current wide-spread use and popularity of TCP to construct distributed applications, it is vital that transparent techniques be developed that allow a client to receive uninterrupted and undegraded services despite server failures. Fault tolerance support at the TCP layer can be very advantageous, and can prove to be more effective than at the application layer.

1.1 Motivations for Client-Transparent Fault-Tolerance

To enable fault tolerance in a TCP based client-server application, cooperation of the client in implementing server fault tolerance is usually required. Here we describe the reasons why client-transparent fault tolerance is very advantageous.

- **No changes required at the client.** Server applications are usually hosted on machines which are under the control of the organization offering those services to the clients. Thus, installing SW, modifying the operating system on these machines is feasible. In contrast, clients are typically widely spread out, numerous, and usually not under the control of any one organization. Installing SW on all the clients is therefore a huge undertaking and in a lot of cases simply not possible.

- **No upgrades. No backwards compatibility issues.** If fault-tolerance code is present on both servers and clients, it is likely that any enhancements or other changes made to the server-side of the SW would require corresponding changes on the client side. This would require updates to be propagated to all the clients. Furthermore, since there will always be clients that do not upgrade, the server

would have to make sure that it is backwards compatible with older versions of the client application (doing otherwise is likely to be unpopular with the users, and could potentially cause users to switch to a competitor). This requirement increases complexity of the server-side code. Clearly, it is an advantage if no code changes at all are required at the client, irrespective of the fault-tolerance mechanism being used at the server.

- **Interoperability for application protocols with no built-in fault-tolerance.** A lot of standardized distributed application level protocols such as FTP, HTTP, etc., which use TCP, do not have any fault-tolerance mechanism defined in the protocol which would allow a user session to be migrated to a backup machine. Thus, any mechanism devised to attain such fault-tolerance is non-standard or even proprietary. This limits the use of such mechanisms, e.g., close cooperation between vendors is required for client-side and server-side implementations of different vendors to interoperate. On the other hand, as long as only server changes are required, even if they are non-standard, there are no interoperability issues.

- **Faster recovery.** Since no re-establishment of the TCP connection, or, re-synchronization of session state between the backup and the client, is required, a client-transparent mechanism is likely to lead to faster failover and recovery, and, may be more suitable for critical applications where fast response times are required.

- **Large failure detection times.** Typically a client connecting to a data center is across a WAN connection. In the paradigm where a client detects server failure, heartbeat messages are usually used. However, fast failure detection over a WAN using such messages is problematic since it is prone to false positives. Furthermore, it adds to the traffic on a already limited bandwidth WAN con-

nection. On the other hand, failure detection at the server end can be made very fast these drawbacks.

If fault-tolerance is supported at the TCP layer (transport layer), clients can be made oblivious to server fault-tolerance. Thus, implementing fault tolerance in server applications (especially ones that have already been deployed in the field), with fault tolerance support at the TCP layer, is more effective: **no fault-tolerance related changes are ever required on the clients**.

1.2 Main challenges

There are a number of challenges related to providing client-transparent server fault tolerance for applications that use TCP connections for communication.

1.2.1 Transport layer challenges

Migrating a TCP connection to a different host, or, even restarting a torn down TCP connection on the same server without client assistance is difficult.

TCP connections were designed to be point to point, and once a TCP connection is established, its two endpoints are fixed with respect to the IP addresses and port numbers used. There is no provision for migration of the connection for server fault tolerance or user mobility.

Migrating a TCP connection is hard since there is a lot of connection-related state information kept in the OS kernel. The most critical part of this state information is segments in the TCP buffers and sequence numbers information. There are also additional parameters such as window sizes, etc.

The segments in the TCP buffers may not be stored anywhere else, and thus, if the TCP connection gets torn down due to a failure, these segments are unrecoverably lost. These are: (1) segments in the TCP receive buffer that have been acked but not

read by the application yet; (2) segments in the TCP send buffer that have not been sent out yet; (3) segments in the TCP send buffer that have been sent out but not received by the client yet.

It is usually hard to dynamically synchronize this state information with a backup without causing server performance to suffer and to deviate from standard TCP behavior. Furthermore, since this state information resides in the OS kernel, kernel level modifications are required.

Client-transparent migration of a TCP connection is harder since it is more restrictive as the new TCP server endpoint cannot renegotiate with the client; the new TCP endpoint must use the same IP address, port number and other connection parameters as the original connection.

1.2.2 Application layer challenges

In order for a backup to provide service to a client, in addition to failover of the TCP connection, the backup application must synchronize with two entities: (1) the primary server application state information before failure, to determine which client request to execute/re-execute; (2) the backup TCP layer, to determine which exact byte to start writing to the TCP socket from.

These application layer issues disappear – indeed, server fault tolerance can be make transparent to the application layer – if the application is deterministic, or, can easily be made so. By a deterministic application, we mean an application whose multiple instances, if given the same input data stream, produce exactly the same output. A deterministic application on the primary server can be kept synchronized with a replica of the application on a backup by replicating the client bytes to the backup. This does not, however, work for non-deterministic applications.

While designing such a fault-tolerant solution for a non-deterministic application, three main application level challenges need to be considered: (1) failover time;

(2) application level modifications; (3) overhead during normal operation (failure-free periods).

Failover time. For any application, a fast failover to a backup application in the event of failure is important, and is more so in the case of an interactive application. Fast failover execution requires that either the state information at a backup application is synchronized with the primary at the time of failure, or, very soon thereafter. For example, a common strategy that involves replaying of all/most client requests at a backup for state synchronization is not acceptable if a fast failover is desired.

Application level modifications. Another challenge is how to keep the changes to an application to a minimum; furthermore, how to abstract out common changes so that they are not implemented separately for each application.

Overhead during normal processing. Excessive overhead during normal processing can degrade server performance. Saving state information on a backup, or, on a shared disk, can cause a excessive overhead if an application has a lot of state information that changes rapidly. For instance, Samba servers keep a substantial amount of state information for each open file. In such cases, trade-offs need to be made between the amount of overhead, failover time and the degree of synchronization depending on the type of application.

1.2.3 Some Examples

We show, through some examples, situations where client transparent fault-tolerance proves to be very useful.

1.2.3.1 HTTP/FTP

Figure 1.2 shows a client downloading a large file using HTTP protocol (or HTTP over TLS) from a server. If the server fails in the midst of the transfer, the client would have to re-issue the request and the file transfer would re-start from the beginning.

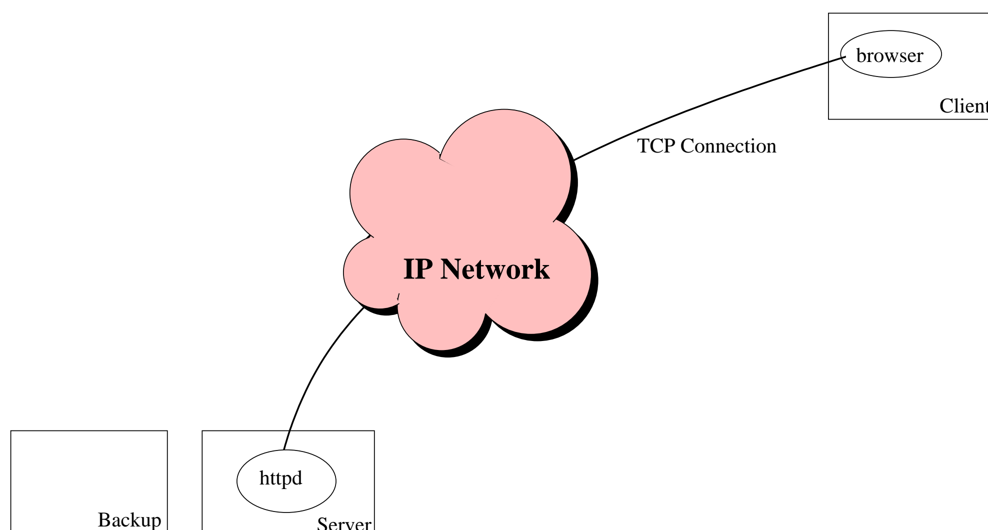


Figure 1.2: A user downloading a file using a browser.

However, if a client-transparent TCP connection migration mechanism is in place, the HTTP session will be migrated to a backup and the user will at most experience a glitch in the service.

This example applies to other TCP based protocols for transferring files as well such as FTP.

1.2.3.2 CIFS/SMB file server

Samba [64] is an open source, Unix-based application that allows Microsoft windows clients to connect to filesystems and printers hosted on Unix machines. It implements the Common Internet File system (CIFS) [20] , which is Microsoft's network file access protocol. CIFS is the successor of the server message block (SMB) protocol (from which Samba derives its name). Samba's main attraction is that it allows sharing of files and printers between unix and windows machines. Furthermore, it provides a cheaper alternative to MS windows servers.

Usually clients connect to the Samba server using TCP. These are long-running TCP sessions. Unlike some other network file access protocols like NFS [65], CIFS

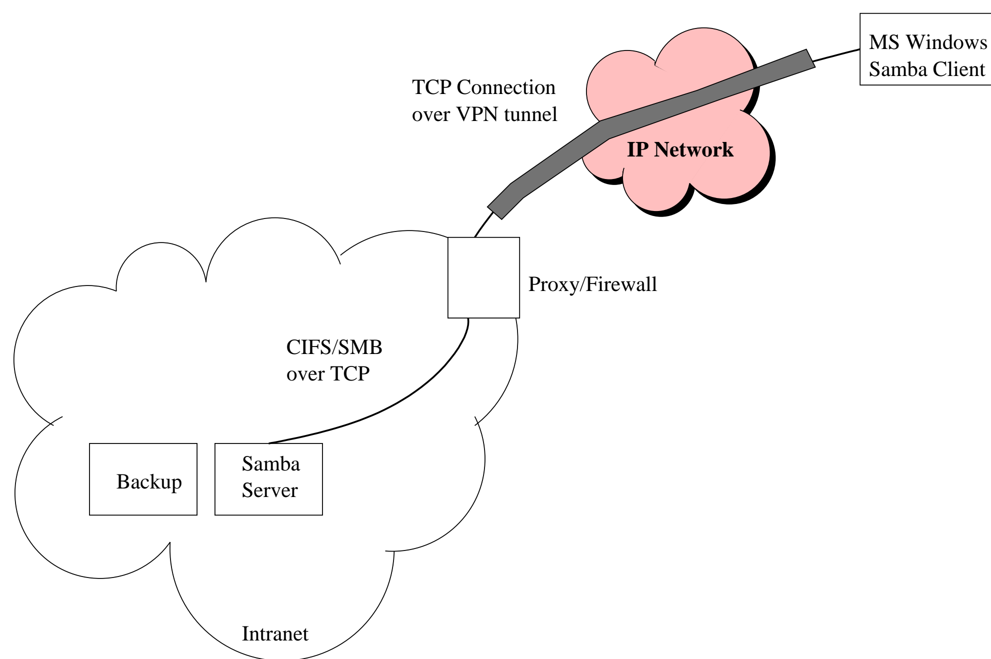


Figure 1.3: A Samba client connecting to a Samba server remotely over a VPN tunnel.

maintains a lot of state information for each open file at the server.

Currently, Samba implementations require the client to reconnect if the server fails. Also, requests being processed at the time of failure are aborted and others in transit may be lost. In fact, documentation on HP's HA (highly-available) CIFS server implementation that is based on Samba warns that [33], "file locks are not preserved during failover," and further that, "if a failover occurs when a print job is in process, the job may be printed twice or not at all, depending on the job state at the time of the failover."

Client transparent fault-tolerance will solve these problems.

Figure 1.3 shows a Samba client connecting over a VPN tunnel to a remotely located Samba server.

1.2.3.3 Media Applications

Usually, UDP is preferred to TCP for real-time media streams since for media "timeliness" is more important than reliable delivery. Also, with TCP, there can be substantial fluctuations in the transmission rate. However, many researchers have argued in support of using TCP, with slight modifications, for media due to its desirable features like congestion avoidance and control, and, flow control, which is not available in UDP [76, 34]. Furthermore, a large number of firewalls disallow all UDP traffic while allowing outgoing TCP connections, which has resulted in TCP being routinely used for media.

In media applications fast failover is critical, and in applications where TCP is used, client transparent failover will prove useful in providing seamless service to the user.

1.2.4 Desirable System Features

We discuss some desirable attributes of a client-transparent server fault tolerant system. We have tried to incorporate these attributes in the fault-tolerance mechanisms proposed in this dissertation.

- **Short Failover time:** To minimize disruptions and maintain continuity of service for the user, the failover time should be short enough so that the user does not experience more than a glitch in service due to the failure. For some applications, e.g. real-time media, it is more critical that the failover time be low than for other applications. There may be some trade-offs involved between a fast failover, and: (1) overhead during normal operation, (2) mechanism and degree of synchronization between the primary and the backup. An application developer, incorporating such a fault-tolerance mechanism into an application should have the flexibility of making these trade-offs.

We recommend that for short failover times, failure detection must be done at the server end and not mainly by a client.

- **Support for non-deterministic applications:** Client-transparent systems such as FT-TCP [4], and [38], require that the server application be deterministic. While a few real-life applications may be deterministic, and, some others that can be easily converted to one, there are numerous others that are not. Thus, accommodating non-deterministic applications should be an important goal for any such fault tolerance mechanism.

- **Multiple backups:** An off-the-shelf backup machine, although reasonably reliable, may not meet the reliability standards set by an organization for an application server. A custom-made machine may meet those standards, however, it is likely to be very expensive. An alternate is to use multiple backups. Multiple backups lead to higher reliability even when the machines individually are not very reliable. Thus, a multiple backup configuration (made up of inexpensive, off-the-shelf machines) can provide a highly reliable and cost effective solution.

- **Geographical Redundancy** Critical applications must continue to provide service under all circumstances. This includes natural and man-made disasters where an entire location is affected, for example, floods, earthquakes, war, terrorist attacks, etc. In order to provide service under these circumstances, a backup server must be available at a geographically separated location from the primary.

- **No single point of failure:** Any well-designed system with fault-tolerance will be able to tolerate single failures. Even when components (e.g. servers) are duplicated, a system designer should ensure that a single failure does not cause

the component pair to fail, e.g., a single power outage should not cause both servers of a pair to lose power.

- **Security:** A fault-tolerance mechanism incorporated in a server application should support existing security mechanism in place.
- **Scalability:** A fault-tolerance mechanism should support the existing scalability techniques in an application.
- **Minimal overhead during normal operation:** During normal operation (failure-free periods), it is desirable that a fault-tolerance mechanism causes minimal overhead.
- **Minimal deviation from standard TCP during normal operation**
- **Minimal changes to the OS kernel**
- **Simplicity of architecture and design**

1.3 Contributions

This thesis postulates that fast, seamless and session-preserving failover, preferably client-transparent, on server failure, is essential to improving user experience. To this end, we have built ST-TCP (Server fault-Tolerant TCP) and CRAFT (Client tRAnsparent Fault-Tolerance). These systems provide infrastructure services to enable applications to provide seamless server fault-tolerance. We have also designed an IP telephony call server fault-tolerance mechanism that seamlessly migrates IP telephony calls on server failure.

1.3.1 ST-TCP

ST-TCP [47, 48] is a primary-backup system with an active backup. The backup taps the traffic between the client and the primary and delivers it to a replica of the

primary application running on the backup. The backup creates a TCP socket identical to the primary's such that it can take over the client-primary TCP connection on failure of the primary. ST-TCP assumes that the primary application is deterministic.

The primary makes sure that the backup has received a particular byte from the client before discarding it from its buffer. It uses extra TCP receive buffer space for this purpose and purges clients bytes from its receive buffers only after a notification from the backup of their successful receipt.

A heartbeat (HB) mechanism exists between the primary and the backup for detecting failures. When the backup detects that the primary has failed, it stops suppressing the output segments that it generates for the client and takes over the client-primary TCP connection. The failover appears seamless to the client since it takes a very short time and the same IP address, port number and sequence numbers are used by the backup.

The specific contributions of ST-TCP are listed below.

- ST-TCP provides client-transparent server fault-tolerance to deterministic applications with (1) no changes required to the application, (2) no deviation from standard TCP behavior, (3) minimal overhead during normal operation, and, (4) seamless, fast failover. None of the other approaches, with similar goals as our research [4, 80, 38, 1], have all of these desirable features.

- The above properties of ST-TCP were demonstrated by conducting experiments using a variety of synthetic applications that mirror the behavior of commonly used applications.

- To address the issue of the backup losing client bytes due to temporary, local failures, a mechanism for retrieving these bytes from the primary is proposed.

1.3.2 CRAFT

CRAFT (Client tRAnsparent Fault-Tolerance) aims to provide the infrastructure necessary to build a flexible, scalable and fault-tolerant web server architecture. It addresses the failure recovery requirements of the *newly emerging* web service applications that are replacing the corresponding desktop applications. These applications have two distinct features that are not present in traditional web service applications: (1) They maintain relatively long-duration, stateful client-server sessions; and (2) The amount of data flow from the clients to the servers is relatively large. As users become dependent on such services in increasing numbers and use them for critical tasks, a failure of servers hosting such applications can cause significant disruption. For a seamless user experience during server failures, it is not sufficient to simply provide server fault-tolerance, e.g., by using data replication and alternate servers. The **quality** of server fault-tolerance, in terms of the impact of a failure on a user, is also very important. Furthermore, as seen by the user, the recovery time must not be more than at most a few seconds.

We architected CRAFT in two phases. The first phase of CRAFT involved enhancements to TCP splicing, a technique commonly used in web switches/proxies for improving the performance of serving web content. We proposed three generic enhancements to TCP splice: (1) Enable a TCP connection to be simultaneously spliced through multiple machines for higher scalability; (2) Make a spliced connection fault-tolerant to proxy/web switch failures; and (3) Provide flexibility of splitting a TCP splice between a proxy and a backend server for further increasing the scalability of a web server system.

The second phase of CRAFT involved providing seamless fault-tolerance support for a backend server. On server failure, the spliced client connection is re-spliced with an alternate backend server and the application state synchronized with minimal changes required in the application. The main components of the architecture are: (1) a proxy

that provides splicing and re-splicing functionality; (2) a logger that transparently saves requests and responses; (3) transactionalization and tagging of the bytes saved at the logger; (4) adaptive failure detection; (5) synchronization and re-splicing after backend server failure; and (6) handling of non-determinism.

The specific contributions of CRAFT are:

- Distributed TCP splice: the same connection can be spliced at different machines.
- Fault-tolerant TCP splice: a splice can survive the failure of the machine that created the splice.
- Split TCP splice: splitting of a single TCP splice into two unidirectional splices with segments in the two directions spliced at different machines.
- Re-splice: Adaptation of TCP splice to survive a backend server failure by undoing and then redoing the server half of the splice with an alternate backend server. This requires the assistance of a logger that can provide the byte offsets of the TCP connection at failure.
- Logger: efficient design of a kernel level logger that transparently logs segments belonging to a TCP connection.
- Transactionalization and tagging of the client–server TCP stream to enable recovery by re-start of failed requests.
- A Linux prototype of the CRAFT system
- Deployment and performance evaluation of our architecture with a real-life application: RoundCube Webmail

1.3.3 Migration of IP Telephony Calls

IP telephony has seen tremendous growth recently, and is replacing circuit-switched telephony systems. Hence, IP telephony components must provide service with high-availability, comparable to traditional telephony systems. In this work, we developed an efficient and scalable fault-tolerance mechanism for migrating calls to an alternate IP telephony call controller (server) in the event of the failure of a call controller or network connectivity to it. The basic idea is to opaquely save call state information in the clients (IP phones, media gateways), similar in concept to how a web server saves HTTP cookies, and on failure as the clients migrate to a new call server, this information is pulled by the new server to reconstruct calls. Furthermore, we developed an efficient algorithm for merging components of migrated calls (which may failover to a call server at different times), so that the same call features are available on the reconstructed calls as the original calls. Some of these techniques have been incorporated into commercial products, resulting in improved fault-tolerance.

The specific contributions of this work are:

- Efficient and scalable mechanism to save call session state at a client for the purpose of providing enhanced server fault-tolerance.
- Merging of call components of a call that failover to an alternate call server at different times.

1.4 Organization

This dissertation is organized as follows. The next section discusses related work. ST-TCP is described in detail in Chapter 3. Chapters 4 and 5 describe CRAFT. This is followed by a Chapter on using our logger for providing transactional semantics to a network interface. Chapter 7 provides a description of our work on migration of IP telephony calls. Finally, conclusions and future work are presented in Chapter 8.

Chapter 2

Related Work

In this section, we provide a description of related research. There has been a lot of research done in the area of fault-tolerance of servers. We focus on transport layer techniques which provide survivability to TCP connections despite server failures. This is challenging since fault-tolerance is not designed into TCP. We also briefly discuss primary-backup and group communication systems.

2.1 Primary-backup Systems

Primary backup systems provide fault-tolerance capabilities by replicating service state on one or more backup servers. Clients interact with the primary server. Backup servers monitor the health of the primary, and in case of a primary server failure, one of the backup server is promoted to act as the new primary server (*failover*). Primary backup techniques have been used to build numerous dependable systems. For example, see [59, 17, 82]. Again, most of these systems require clients to be aware of the protocols being used, and maintain the identity of the server. On a failover, the new primary and the client re-establish the connection. This usually requires an application level protocol which must be built into the client as well as the server. ST-TCP is a primary-backup system which is client transparent. Some other such systems are FT-TCP [80], [1], and, [38].

2.2 Transport Layer Techniques

There are approaches for achieving fault tolerance aimed at the transport-level protocols such as TCP. These include FT-TCP [4], HydraNet [68], [55], M-TCP [73], [38], [1], and SCTP [71]. ST-TCP and CRAFT belong to this category.

2.2.1 FT-TCP

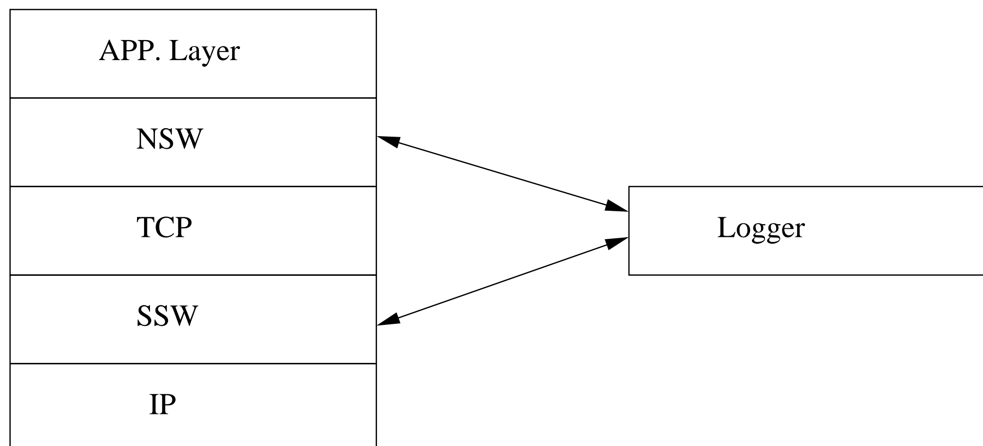


Figure 2.1: Wrappers in FT-TCP.

In FT-TCP[4], server fault tolerance in TCP is provided by using the standard primary backup approach. Two wrappers (SSW and NSW), see Figure 2.1, are put around the TCP server code to forward TCP byte stream to a logger, and ensure that the server state stored at the logger is consistent with the primary server state. In a later work [80], instead of sending the byte stream to a logger, it is sent to an active backup.

In case of a server failure, a new server is started using the state stored in the logger. During this failover period, the client TCP connection is kept alive by sending zero-size window advertisements at regular intervals. ST-TCP improves on FT-TCP by providing a fast failover. The failover time in FT-TCP can be fairly large. This is

because a failover in FT-TCP requires failure detection, time for the backup server to start, and time to update the backup server state from all the data saved in the logger (which could be quite large for long running applications). Thus, although the TCP connection does not break here, a client will certainly see a disruption and degradation in service. ST-TCP, on the other hand provides a very fast failover. Note that FT-TCP could in principle be used with active replication and in this way could reduce the fail-over time.

ST-TCP makes different trade-offs compared to FT-TCP. ST-TCP does not only optimize the fail-over time but also the failure-free case. By tapping the packets, the backup does not increase the latency nor does it decrease the bandwidth.

Second, even during failure-free periods, standard TCP behavior can be affected in FT-TCP in the following way. Acknowledgments sent by the TCP server are delayed by SSW until the relevant TCP bytes are logged in the logger. In addition, SSW does extra processing to recompute checksums and map sequence numbers. These delays would cause a client and server to make a higher estimate of the round trip time. Also, writes by an application on the server side may be delayed by NSW until NSW receives appropriate acknowledgments from the logger. In ST-TCP, no changes in the standard TCP behavior occur during failure-free period. Finally, the important issue of what happens if a TCP server is wrongly detected to have failed has not been discussed in [4]. ST-TCP specifically addresses this and other failure scenarios.

In [80], the authors report running two real-life applications over FT-TCP. These are: (1) a Samba server, and, (2) Darwin Stream Server. To decrease failover time, FT-TCP is run with an active backup instead of a logger. The sources of non-determinism in these applications are minor and thus it was easy to add code to synchronize the primary and the backup application in those instances.

2.2.2 HydraNet

HydraNet-FT [68] is an infrastructure to dynamically replicate services across an internetwork and have replicas provide a single, fault-tolerant service access point to clients. The service access point is facilitated by an IP-redirector that provides one-to-many message delivery to replicas, and many-to-one message delivery from the replicas to the client. ST-TCP is based on a similar idea of redirecting TCP byte stream to backup servers, in addition to the primary server. The main difference between HydraNet-FT and ST-TCP is that the process of redirection is extremely simple (tapping TCP byte stream) in ST-TCP. In fact, there is no separate redirector component in ST-TCP. The main reason for this simplicity is that while ST-TCP is focused on providing server fault-tolerance in TCP, HydraNet-FT aims to provide additional services such as replica management and load balancing. As a result, the design of redirector is complicated.

2.2.3 Orgiyan and Fetzer

Another approach for providing fault tolerance at the TCP layer is discussed in [55]. A backup server is used to tap the Ethernet to read all the packets destined for the primary. A layer is added between the TCP and the application layer on the servers as well as the client. When the TCP connection breaks or the primary fails, this layer in the backup server and the client establish a new TCP connection. Although, this makes a connection failure transparent to the client application, it does require modifications on the client machine (installing a client wrapper). This defeats our main goal of not requiring any changes on the client end.

2.2.4 M-TCP

A recent work addressing continuity of services implemented using TCP is M-TCP (Migratory TCP) [73]. M-TCP operates in an environment where a service is

implemented by a pool of servers. A client establishes a TCP connection with one of the servers. Each server maintains a fine-grained checkpoint of each TCP connection state. TCP connection migration from its current server S_c to the another server S_n in the server pool depends on the migration policy. For example, it can be initiated by the client when it notices that its service has deteriorated. At this point, S_c transfers the checkpointed state C_p of this connection to S_n, S_n starts a new TCP connection and initializes its state using C_p, and then sends a message to the client to continue its operation.

ST-TCP is different from M-TCP in the following ways. First, M-TCP requires clients to monitor the health of its server and initiate connection migration. This is done by installing a wrapper on the client machine. ST-TCP is completely transparent to the client (not even a client wrapper is needed). Second, M-TCP is designed to tolerate server overload and network congestion between client and server. It cannot tolerate server failure. ST-TCP on the other hand provides continued service despite server failure. Finally, the migration period can become large in M-TCP, as it involves detecting service deterioration, transfer of connection state, and starting a new connection. As mentioned earlier, ST-TCP provides a very fast failover.

2.2.5 TCP Migrate

TCP migration [69] is a technique that is transparent to the client application but requires modifications to both the client and server TCP/IP stacks. Modifications to the network infrastructure (e.g., Internet routers, underlying protocols) are not required. The client or any of the servers can initiate migration of the connection. At any point in time, only one server is connected to the client.

2.2.6 SCTP

SCTP [71] provides support for tolerating network congestion and failure by establishing multiple redundant paths between the client and the server. It supports use of multi-homed servers, where each server NIC could potentially be connected to a different service provider. Although, SCTP connections can be seamlessly migrated from one server NIC to another, it does not tolerate server failures. We note that ST-TCP can be extended to operate in conjunction with SCTP. This will result in a more powerful transport-level service that can tolerate both network failures/congestion and server failures.

2.2.7 Swift

The SwiFT system [35] provides fault tolerance for user applications. SwiFT consists of modules for error detection and recovery, checkpointing, event logging and replay, communication error recovery and IP packet rerouting. The latter is achieved by providing a single IP image for a cluster of server computers. Addressing within the cluster is done by MAC addresses. All traffic from the clients is sent to a dispatcher, which forwards the packets to one of the server computers. A client must run the SwiFT client software to reestablish the TCP connection if the server fails.

2.2.8 TCP Splice

TCP splicing [44, 70] is a technique that is used to improve performance and scalability of application-level gateways. Clients establish TCP connections to a dispatcher application. The dispatcher chooses an appropriate server to handle a client connection, and then modifies the TCP/IP stack of the dispatcher to forward all TCP packets of the connection directly to the selected server. No further involvement of the dispatcher is required until the connection is terminated. TCP splicing requires all traffic to flow through the dispatcher. We use TCP splicing at a proxy in CRAFT for client transpar-

ent mobility of TCP connections from a primary server to a backup in the event of the primary's failure.

2.2.9 Backdoors

Backdoors [16, 72] is designed for client transparent server fault-tolerance. In addition, it also handles non-deterministic applications. These design goals are very similar to those of CRAFT. Backdoors does not replicate state information of the primary server, but assumes that in most server crashes, the memory remains intact. It uses a specialized network interface card (from Myrinet) to remotely access the memory of the failed machine to obtain all the state information required to migrate the client session to a backup machine. This state information consists of both TCP state information in the kernel and application specific session state information. Backdoors does not support failures that could corrupt memory, or those that could preclude access to the memory of the failed machine. In comparison, CRAFT actively replicates transport and application state information and does not in any way depend on the failed machine. Consequently, any failures of the primary server, including ones that may damage a primary physically (e.g. floods, fire, etc.) are tolerated by CRAFT but may not be tolerated by backdoors.

2.3 Group Communication

Group communication systems comprise of a set of fault-tolerant protocols designed to provide support for object replication. Examples include [51, 6, 5, 78, 52, 15]. While group communication systems have proved to be useful for constructing dependable distributed applications, they are difficult to deploy for large-scale applications. This is because they rely on their own programming interfaces, which must be used in both server and client programs, thus making the integration of an existing client base difficult.

Chapter 3

Server fault-Tolerant TCP

In this section, we describe the design, implementation, and performance evaluation of ST-TCP (Server fault-Tolerant TCP), which is an extension of TCP to tolerate TCP server failures.

ST-TCP uses an active backup server that keeps track of the state of the TCP connection and takes over the TCP connection when the primary fails. This migration of the TCP connection to the backup is completely transparent to the client. Because no changes are required on the client machine, any TCP client can access a ST-TCP server. The performance overhead of ST-TCP over standard TCP is minimal, and during normal operation its behavior is the same as that of a regular TCP. In addition, ST-TCP provides a fast and seamless failover whenever the primary server fails. This is verified by a prototype implementation of ST-TCP in the Linux operating system, and experiments with a number of simulated applications which have different communication characteristics.

3.1 Requirements

There are five important goals in the design and implementation of ST-TCP. The first and the most important goal is that ST-TCP should be able to support existing TCP-based client server applications without requiring any changes in the client application or server application code. This will allow an easy transition for existing client

server applications to become server fault-tolerant. The second goal is that ST-TCP should not require clients to install any special software (e.g., wrappers or other library functions) on their machine. The main reason for this is that clients are typically distributed all over and not centrally controlled, thus making it extremely difficult to upgrade all of them. Also, depending on the administrative policies, a client may not be allowed to install, or discouraged from installing custom-software such as ST-TCP. On the other hand, changes are easier to make on the server side, since servers are typically less in number, and in control of the organization providing the service. These two goals require that server fault tolerance should be completely transparent to client application, client-side TCP, client machine, and server application.

The third goal is that ST-TCP should not have any effect on the client server communication during failure-free periods. In particular, there should not be any effect of the fault-tolerance technique incorporated in TCP on the TCP performance during failure-free periods. This is important, because any degradation in TCP performance, however small that may be, can significantly affect an application's performance. This is especially true for the common case where server failures are quite infrequent. The fourth goal is that the failover period of ST-TCP should be fairly short. It should be short enough that the clients will barely notice a change in the service being provided. This is important, because most application clients will simply break a TCP connection on their own if the failover period happens to be large. This is despite the fact that the client TCP itself may not break the connection during that time.

Finally, the technique used to provide server fault tolerance should be simple and easy to implement, and should require minimal changes on the server side. The main reason for this is that any complex scheme that requires extensive changes on the server side will be psychologically unacceptable to the administration managing a server. So, installation of a large number of new library routines on the server side, or a massive overhaul of TCP server code is out of question for ST-TCP.

3.2 Architecture Overview

ST-TCP is based on the idea of primary/backup approach to provide high availability and fault tolerance. To ensure fast failover, ST-TCP maintains active backup servers that can take over the functions of the primary server as soon as any failure of the primary server is detected. To avoid any performance overhead during failure-free periods, ST-TCP requires only a few changes on the server-side TCP that do not affect the normal flow of TCP byte stream.

The main idea in the design of ST-TCP is that one or more backup servers receive the TCP segments exchanged between a client and a primary server by simply tapping the TCP byte stream at an intermediate point in the connection between the client and the server. Figure 3.1 illustrates this idea. Since every TCP segment exchanged between the client and the primary server is tapped in by the backup server, the backup server learns the complete communication state of the primary server. In particular, we can guarantee that a non-crashed backup server receives all data that the primary server receives. In addition, it also receives all data that the client receives.

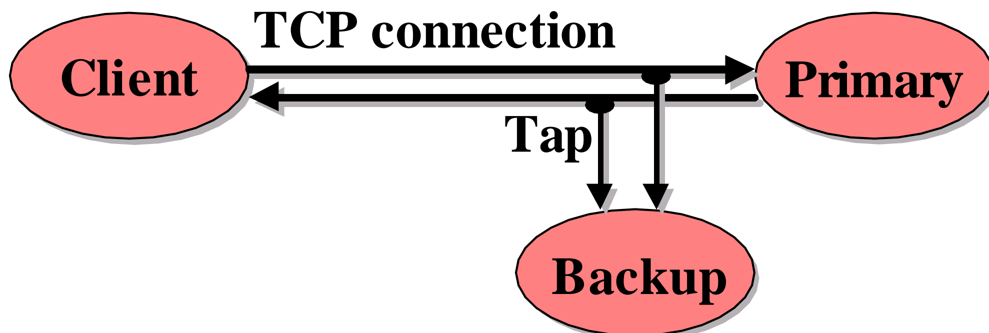


Figure 3.1: Tapping TCP byte stream.

In case a server is completely deterministic, a backup server can keep its state consistent with the state of the primary server by executing the same sequence of requests as the primary server does. Many servers are however non-deterministic, e.g.,

timestamps or ids used by the backup server may not be exactly the same as those used by the primary server. Replication of non-deterministic servers has been researched extensively. To keep the primary and backup servers in sync, one can combine this protocol with a *leader/follower* consistency protocol (e.g., [13]). In what follows, we assume that applications are deterministic or that they have been made deterministic, e.g. by use of a leader/follower protocol.

A backup server can detect the failure of a primary server when its state becomes inconsistent with, or its (suppressed) replies differ from, that of the primary server. This allows a backup server to **detect** failures of the primary server that are different from crash or performance failures. Crash and performance failures of the primary server are detected by the backup using a simple timeout mechanism. For this reason, the primary server sends periodic heartbeat messages to the backup server. We enforce consistent behavior by making sure that the primary server is never incorrectly suspected to have failed (see Section 3.2.2).

3.2.1 Ethernet Tapping

Most local area networks are Ethernet based. Therefore, we assume that servers communicate with clients via a local area Ethernet. The clients are connected to the Ethernet by one or more gateways. This means that all TCP traffic between the server and the clients can be tapped from the local Ethernet.

Ethernet has originally been a broadcast medium. Broadcast media simplify the tapping of communication streams. [55] discusses different tapping architectures for broadcast Ethernet. However, in recent years most Ethernet installations have been converted to switched Ethernet (see Figure 3.2). Logically, an Ethernet switch replaces the broadcast medium by a crossbar. This prevents a backup node from tapping the traffic of the primary node since a node only receives packets addressed to itself.

Some managed Ethernet switches provide an option to forward traffic flowing

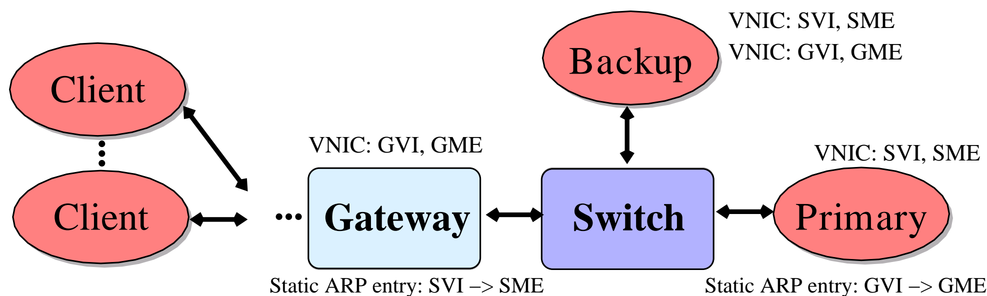


Figure 3.2: Ethernet switch tapping architecture.

from/to a port to some other port. This feature permits us to sniff all traffic between the server and the clients. To facilitate tapping in switched Ethernet without this capability, we also investigated a solution that maps a unicast IP address to a multicast Ethernet address to ensure that all packets received or sent by the primary are forwarded by the Ethernet switch to the backup.

The service provided by the primary can be accessed by the clients via a static virtual IP address, SVI. We create at the backup and the primary node a virtual network interface (VNIC), i.e., a software NIC mapped to a hardware NIC. We assign SVI to these VNICs. Further, we assign a fixed multicast Ethernet address SME to these VNICs on the primary and backup machines. The service IP address SVI is statically mapped onto the multicast address SME in the gateway ARP table. This static mapping is necessary since the IPv4 router requirements RFC [12] disallows an IP router to accept a multicast link layer address in an ARP reply. The mapping of SVI to SME permits the backup node to tap all traffic from the clients to the server.

To tap the traffic from the primary server to the clients, we similarly create VNICs in the gateway and backup nodes and assign each with a static virtual IP address (GVI) and a fixed multicast Ethernet address (GME). A static GVI to GME mapping is created in the primary ARP table. The backup node can in this way tap all traffic from the primary to the client. Note that most gateway machines run modern operating

systems like Linux, because they often provide additional services like firewall and/or VPN functionality. Hence, we can install VNICs on these gateways.

3.2.2 System Architecture

Dependable systems often have the requirement of having no single point of failure. Avoiding a single point of failure can increase the availability of a system and simplifies the replacement of components. ST-TCP supports system architecture without a single point of failure like the one depicted in Figure 3.3. All components are replicated. Primary and backup are connected to two switches that are connected via two loggers to two gateways. One can use the two links to increase the bandwidth. In particular, for full-duplex Ethernet links to the server one would configure ST-TCP such that the backup receives the packets to and from the server on two separate Ethernet links.

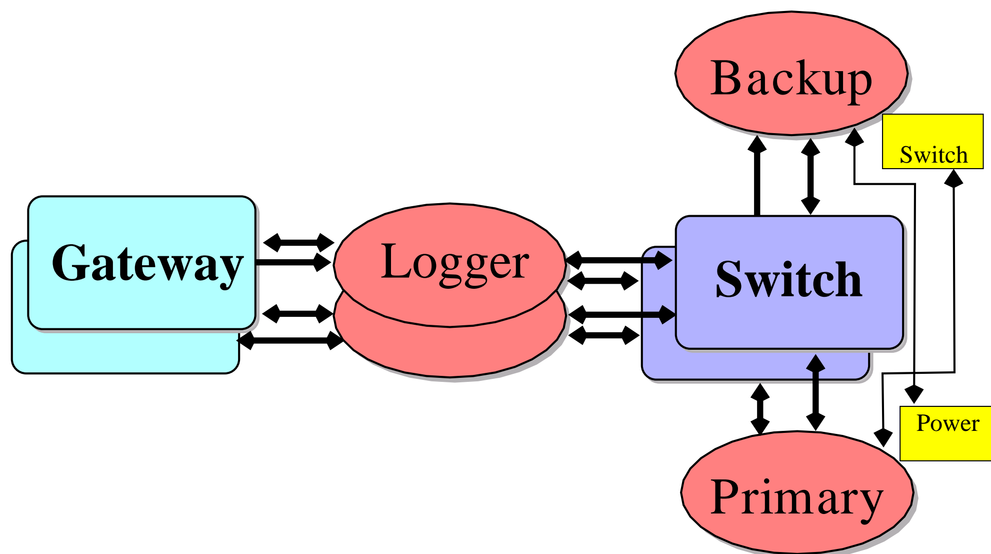


Figure 3.3: System Architecture without a single point of failure.

For ST-TCP to work correctly, we need a perfect failure detector. In particular, the backup is never permitted to suspect the primary if the primary is still up. To do

that, we can use the perfect failure detector protocol of [26] that was designed for this situation. Alternatively, we can use controllable power switches. If the backup suspects the primary, it switches off the power of the primary. This makes sure that the primary is crashed before the backup takes over the IP address of the service.

As we will describe in Section 3.3, the backup asks the primary for packets that it failed to receive from the Ethernet (but were received by the primary). In case such an omission failure happens together with a crash of the primary, the backup can in certain cases not take over as primary because the complete communication state is not known to the backup. To mask such double failures, one can insert a logger into the network [55]. This logger machine logs all packets on the Ethernet in its main memory for a bounded amount of time. Since the backup will suspect the primary within a bounded amount of time after the primary crashed, the backup can recover all missing packets from the logger.

The logger introduces a very small delay but does not reduce the bandwidth. The memory needed by the logger depends on the maximum bandwidth of the Ethernet and the maximum failover time. Since all traffic to and from the server has to flow through the logger(s), the logger(s) has (have) the complete communication state. By having two loggers, or a bypass network for a failed logger, one can prevent the logger from becoming a single point of failure.

In addition to avoiding single points of failure, we want to avoid the introduction of performance bottlenecks, i.e., new CPU or bandwidth limitations due to replication. For half-duplex Ethernet one Ethernet NIC has sufficient bandwidth to tap the traffic between the primary and its clients. In full-duplex mode the bandwidth of a single Ethernet NIC might not be sufficient to capture the complete traffic. In the latter case, one can use one Ethernet NIC to tap the traffic for each data direction. Note that due to other bandwidth limitations (e.g., WAN bandwidth limitations) one might not always need additional NICs.

Due to the tapping of the Ethernet packets from the primary to the clients, the backup system needs additional CPU resources. Modern operating systems support the parallel processing of network packets. Hence, the addition of CPUs to the backup system can address CPU limitations introduced by the Ethernet tapping. Note that the addition of CPUs can reduce new CPU bottlenecks even if the application itself is only single threaded.

3.3 Protocol Details

We will describe the protocol details of ST-TCP by focusing on three important design issues: (1) starting the primary and backup servers, (2) operation of the primary and the backup servers during failure-free periods, and (3) failure detection. In this section, we will focus on the protocol design issues. Implementation details are given in Section 3.4.

3.3.0.1 Initialization

The primary and the backup are configured as described in the previous section so that the backup network interface receives all packets destined for the primary. Further, the backup's network interface is setup such that all packets from or to the server are accepted by it and passed to the higher layers in the TCP/IP stack. To ensure that the backup can take over a TCP connection from the primary in case of a primary crash, it needs to shadow the state of that connection, i.e., maintain a consistent state of that TCP connection so that on failover it is indistinguishable from the primary to the client.

The server application is started on both the primary and the backup. Since it is the same application, the same listen port is used on both the machines. To avoid any conflicts, we assume that the backup is running on a dedicated machine, and not used for any other purpose.

In addition to IP addresses and port numbers, another stateful entity in a TCP

connection that the backup must be aware of is the sequence numbers used in that connection. The backup must either use the same sequence numbers as the primary or be able to map its sequence numbers for that connection to those of the primary so that the connection migration works. In ST-TCP, backup uses the same sequence numbers as those used by the primary for a shadowed TCP connection. The steps involved in the initialization of such a TCP connection are described below.

(1) To initiate the TCP connection, the client sends a SYN segment to the primary. The backup also receives this SYN segment.

(2) In response to the SYN, the TCP layer in both the primary and the backup sends a SYN/ACK to the client. This SYN/ACK is dropped (suppressed) on the backup. The primary's SYN/ACK is received by the client.

(3) The client's ACK segment, completing the three way handshake, is used by the backup to modify its own initial sequence number and other variables related to the initial sequence number. After this point, the backup's sequence numbers match those of the primary for this connection.

3.3.1 Failure-free Period

During failure-free period, all TCP segments exchanged between a client and a server are received by the backup server. The backup server executes as a normal TCP server, except that all replies from the backup server to the client are dropped. ST-TCP has incorporated two additional functionalities – (1) ensure that the state of the backup server is consistent with the state of the primary server, and (2) detect failures of (primary or backup) servers.

For simplicity, let us assume that the server application is deterministic. This implies that the (suppressed) replies of the backup server will be identical to the replies of the primary server, as long as the backup server receives the same sequence of bytes

from the client as the sequence of bytes received by the primary. In general, this will be the case since the same sequence of bytes that are received by the primary server are tapped by the backup server. However, it is possible that some TCP segments, received correctly by the primary, get lost before reaching the backup server. For example, this can happen if the IP stack on the backup server drops IP packets because of an IP-buffer overflow. Since the primary server receives these segments, it will acknowledge these bytes. This will result in the client-side TCP purging those segments from its send buffer. As a result, there is no way for the backup server to retrieve those lost TCP segments.

We address this problem and the issue of server failure detection by establishing an alternate connection between the primary and the backup server, and modifying the TCP buffer management in the primary server. A separate UDP channel is established between the primary and the backup servers when these servers are started. This channel is used to make sure that the backup server receives all TCP segments from the client that are received by the primary server. The backup server uses this channel to send a request to the primary server asking for missing TCP segments, when it discovers that it has missed some TCP segments. This channel is also used by both the primary and the backup servers to monitor each other by sending periodic heartbeat (HB) messages. Implementation details of how this alternate channel is established are given in Section 3.4.

During failure-free periods, the primary server needs to make sure that before it discards a received TCP byte from its receive buffer, the backup server has already received it. So, while in standard TCP, a byte received from the client is discarded once it has been read by the application, ST-TCP requires that this byte, in addition, be acknowledged by the backup server before being discarded. The backup server uses the alternate communication channel to acknowledge (to the primary) the sequence numbers of the bytes it has received from the client. The actual strategy for when and

how this acknowledgment is sent is described later.

Figure 3.4 shows the receive buffer of the primary ST-TCP server. For comparison, we have also included the corresponding receive buffer of a standard TCP server. As shown, the receiver buffer of the primary ST-TCP server maintains an additional pointer (`LastByteAcked`). This pointer is the sequence number of the last byte acknowledged by the backup server to the primary server. `LastByteRead` is the sequence number of the last byte read by the application, `NextByteExpected` is the sequence number of the next byte the TCP server expects to receive, and `LastByteRecd` is the sequence number of the last byte received by the TCP server.

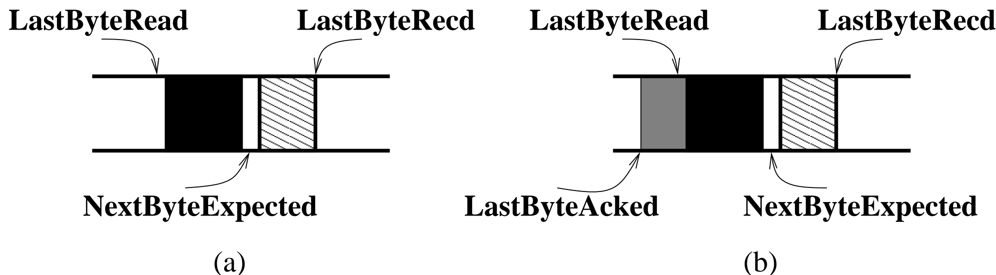


Figure 3.4: Receive buffer of TCP server: (a) Standard TCP, (b) ST-TCP.

In general, the `LastByteAcked` can be smaller than, equal to, or larger than the `LastByteRead`. If it is smaller than `LastByteRead`, all bytes whose sequence numbers fall between it and `LastByteRead` are the bytes that a standard TCP server would have discarded, but a primary ST-TCP server does not discard. A primary ST-TCP server discards all those bytes whose sequence numbers are smaller than or equal to `LastByteRead` or `LastByteAcked`, whichever is smaller. This strategy ensures that any TCP byte that backup fails to receive from its tapped byte stream can be retrieved from the primary server. In general, we expect that the backup server will be able to receive all the bytes from its tapped TCP stream, and there will not be a need for it to request the bytes from the primary server.

Depending on the acknowledgment strategy that the backup server adopts, it is

possible that some bytes will be kept longer in the receive buffer of the primary ST-TCP buffer than a standard TCP buffer. This means that the buffer may fill up sooner in ST-TCP than in standard TCP. Also, this will reduce the advertised window size that the TCP server sends to client. To compensate for this and to ensure that the ST-TCP behavior remains as close to standard TCP behavior as possible, we double the space allocated for the receive buffer. The Berkeley socket interface allows this on a per connection basis.

With more buffer space available for the receive buffer, the extra buffer space can be managed in a number of ways. We use the simplest approach here. The additional space is used only for storing bytes that have not been acknowledged by the backup server but that have been read by the server application. In other words, ST-TCP logically maintains two receive buffers, each with their own limits. The management of the first buffer is identical to the management of the receive buffer in standard TCP, except that some bytes may move to the second buffer before being discarded.

This simple approach ensures that the behavior of the ST-TCP server is identical to the behavior of a standard TCP server from a client's perspective. As long as the backup server keeps sending acknowledgments to the primary server at regular intervals, there will be no difference between the standard TCP server and the ST-TCP server as far as the advertised window size, bytes acknowledged, or any TCP timer calculations are concerned. In other words, during failure-free periods, a client will see no difference between a standard TCP and an ST-TCP.

The behavior of ST-TCP will differ from that of standard TCP if the second buffer fills up. This can happen when the backup server is too slow in acknowledging the bytes it has received, or if it misses some bytes from its tapped TCP byte stream and requests a transmission of those bytes from the primary server. We address this by instituting an efficient acknowledgment strategy in the backup server that is described later in this section.

Clearly, there are other approaches for managing the two buffers that might improve the overall performance of TCP. For example, it is possible to store extra bytes in the second buffer and advertise a larger window size, if the first buffer fills up and there is enough space available in the second buffer. Although these approaches may give better performance results in some situations, they are more complex to implement, and at present, they are not considered.

3.3.2 Synchronizing The Backup Server

The primary server waits for acknowledgments from the backup server before deleting corresponding bytes from its second receive buffer. The alternate UDP channel between the primary and the backup servers is used for this purpose. The backup server sends acks over this channel containing a sequence number that is one less than its `NextByteExpected` value. The frequency with which these acks are sent has a significant effect on the ST-TCP performance. If these acks are sent very frequently, e.g., after receiving every byte, the primary server may end up spending too much time in processing these acks, thus affecting the net TCP throughput. On the other hand, if these acks are sent only once in a while, the second receive buffer in the primary server will fill up, again affecting the net TCP throughput.

The backup server in ST-TCP uses a very simple strategy to send these acks. Instead of sending an ack after receiving every byte, the backup server maintains an integer variable, `LastByteAcked`. The value of this variable is the sequence number of the last byte that was acknowledged to the primary server. An ack is sent whenever any of the following events occur (`X` is a configuration parameter):

- `NextByteExpected` $-$ `LastByteAcked` $\geq$ `X`, or
- a fixed time interval `SyncTime` has elapsed since sending the last ack.

The first event corresponds to the condition when the backup server has received

at least `X` bytes (in the correct order) from the client since sending the last acknowledgment. The second event corresponds to the condition when less than `X` bytes (in the correct order) are received from the client in the last `SyncTime` time units, since sending the last acknowledgment.

The value of `X` naturally depends on the size of the second receive buffer in the primary server. As a start, we have chosen to fix `X` as three-fourths the size of the second buffer. For example, if the size of the second buffer if 4 KB, `X` = 3 KB. The value of `SyncTime` depends on how frequently the backup server and the primary server need to monitor each other. We use the acks sent by the backup server and its response sent back by the primary (which also serve as heartbeat messages) as a mechanism to monitor the liveness of the primary and the backup servers. We have experimented with different values of `SyncTime` ranging from 50 milliseconds to 5 seconds.

The extra traffic introduced by the alternate UDP channel is quite insignificant. For example, assume that the total length (including all header overheads down to Ethernet) of an ack packet is 128 bytes, and there is only client traffic on the LAN (worst case). In this case, one ack packet for every 3 KB of client data increases the LAN traffic by only 4.17%.

3.3.3 Failure Detection

We assume that the computers (primary or the backup servers) have crash/performance failure semantics[24]. The failure detection is based on a timeout mechanism. The backup monitors heartbeat (HB) messages from the primary to detect primary's failure and take over its TCP connection.

The primary monitors the HB messages from the backup and the size of its second receive buffer to determine if the backup has failed. On detecting failure of the backup, the primary transitions to non-fault-tolerant mode.

The failure detection mechanism will eventually suspect a crashed computer.

However, it might wrongly suspect non-crashed computers. We convert wrong suspicions into correct suspicions by switching off the power of a suspected computer (see also Section 3.2.2) before propagating the suspicion .

3.4 Implementation

We have implemented a prototype of ST-TCP. This prototype runs on Linux operating system (kernel 2.2.18), and involved modifying the TCP/IP stack in the kernel. On the backup ST-TCP server, changes were made to synchronize the initial sequence number with that of the primary on TCP connection initialization and to discard TCP segments to the client during failure free periods.

Recall that the primary and the backup servers maintain a UDP channel to help the backup server recover from temporary communication failures of the tapped TCP byte stream, and monitor each other by exchanging Heartbeat messages containing sequence number information. There are at least two ways to establish this UDP channel. One way is to establish this channel in the kernel itself, between the backup and the primary. The advantage of this approach is that sequence number data does not have to be copied from the kernel space to user space. However, it is best to avoid changes in the kernel whenever possible, and although this approach is slightly more efficient, it was not followed.

The other approach, which we used, is to implement this channel in a user process outside the kernel. This requires that the user process has access to sequence number data. This is done by using the `/proc` filesystem in Linux which allows the user process to open a file and read this data. In our prototype implementation, when the backup takes over, it sets a flag in the `/proc` filesystem to indicate to the kernel that the backup has taken over. As soon as the flag is set, the kernel starts sending the packets to the client instead of dropping them.

3.5 Performance

To provide a proof of concept, we measured the performance of ST-TCP with simulations of applications representing different communication characteristics. The main focus of these experiments is to show that no changes are required in the client, there is no deviation from standard TCP during failure-free periods, and that the failover from primary to backup is fast.

Three different applications, with differing communication behaviors, are considered. The first application (**Echo**) consists of a client sending a small message (about 150 bytes) to the server, and the server responding back with the same message to the client. The client waits to receive the echo response before issuing another request. One run of this application involves 100 such message exchanges. The second application (**Interactive**) consists of a client sending a small request (about 150 bytes) and the server responding with an appropriate reply consisting of moderate size data (10 KB). One run of this application also involves 100 requests and the corresponding responses. A new request is sent only after the response to the previous one is received. The third application (**Bulk transfer**) consists of a client sending a small request (about 150 bytes) to the server, and the server responding with a large data file. Files sizes of 1 MB, 5 MB, 20 MB and 100 MB are used for this experiment. As an analogy, the communication pattern of **Echo** is similar to the one displayed by `telnet`, the communication pattern of **Interactive** is similar to the one displayed by `http`, and the communication pattern of **Bulk transfer** is similar to the one displayed by `ftp`.

Experimental Setup. We use three machines to run the experiments. The primary and the backup are 800 MHz AMD Athlon PCs, with 512 KB of cache and 256 MB of memory. Both the machines have Linux kernel 2.2.18 installed on them. The client is a 900 MHz Pentium III Laptop also running Linux (although it could be running any OS that provides a TCP/IP stack). All the machines have a 10/100 Mbit

network interface card. These three machines are placed on the same LAN using a 10/100 Mbit Ethernet hub. Since the hub broadcasts all traffic on all ports, the backup can tap into all of the primary's network traffic. Using an Ethernet switch will lead to a higher throughput.

For each of the three applications, we performed two sets of measurements. The first one measures the performance overhead of ST-TCP over standard TCP when there are no failures in the system. As mentioned earlier, an important goal in the design of ST-TCP is to keep this overhead insignificant. This measurement is done by separately running the applications, with standard TCP and with ST-TCP. For ST-TCP the applications are run for varying values of the HB interval. The second set of performance measurement measures the time it takes for the backup server to take over the primary server when the primary server crashes. This is also called the *failover* time. As mentioned earlier, an important goal in the design of ST-TCP is to keep this time fairly short, so that a client will barely notice a disruption in service continuity. For all the applications, the failover time is measured for various HB intervals. All measurements taken were repeated at least three times and their average values were used. Further, the TCP timestamp option was disabled on the primary and the backup during these experiments.

3.5.1 Performance Overhead of ST-TCP

A performance comparison between standard TCP and ST-TCP is shown in Table 3.1 when there are no failures. For all three applications, there is no significant difference between the average time taken with standard TCP and ST-TCP. Furthermore, for ST-TCP, the average time taken does not differ significantly for different values of the HB. This demonstrates that ST-TCP does not incur any performance overhead over the standard TCP.

	Average Total Time (in secs) without failure					
	Echo Application	Interactive Application	Bulk Transfer Application			
			1 MB	5 MB	20 MB	100 MB
Standard TCP	0.892	2.000	0.640	3.199	12.788	63.952
ST-TCP 5s HB	0.889	2.000	0.639	3.196	12.791	63.950
ST-TCP 1s HB	0.890	1.998	0.641	3.200	12.824	63.883
ST-TCP 200ms HB	0.898	2.000	0.640	3.203	12.794	63.964
ST-TCP 50ms HB	0.896	1.998	0.658	3.201	12.897	63.883

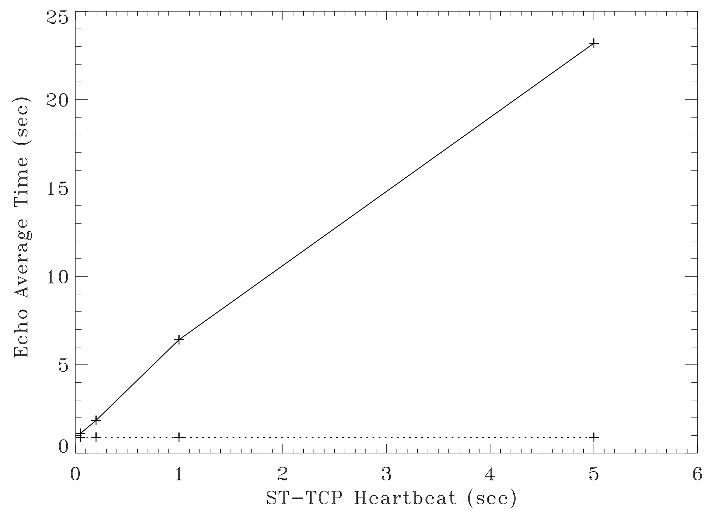
Table 3.1: Comparison of standard TCP with ST-TCP during failure free period.

3.5.2 Failover Time in ST-TCP

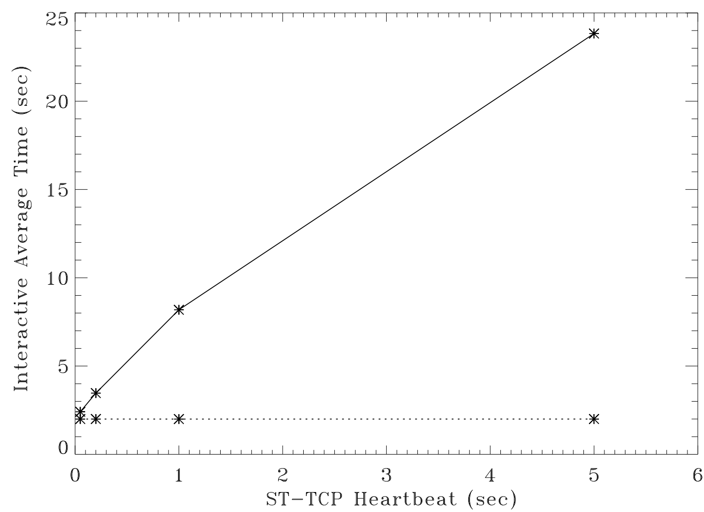
To measure the failover time, a primary crash failure was introduced while the application was running. The failover time depends on two parameters. First, the time it takes for the backup to detect a failure, which directly depends on HB frequency. In our experiments, the backup concluded that the primary has crashed after missing three consecutive HB from the primary. For example, with an HB every 5 sec, the backup will detect primary crash in 15 to 20 seconds depending on when exactly the failure occurs.

The second parameter determining the failover time is the increase in the value the TCP retransmission timeout (RTO) during the time the backup took to detect the failure. This depends on the RTO when the failure occurred and the RTO backoff algorithm. In Linux, the RTO is computed using the round trip time (RTT) and is increased by a factor of two with every retransmission. The lower and upper bound for the RTO in Linux are 200 ms and 2 min respectively.

Figures 3.5.a, 3.5.b and 3.6 show the time taken by one run of each application when there is no failure and in the presence of a failure. The main observation here is that the failover time is directly dependent on the HB interval. The graphs show that the total time taken increases as the HB interval increases. This is because it takes longer to detect failure if the HB interval is large. In Figures 3.5.a and 3.5.b the upper curve depicts the failure case. The lower curve shows the failure free case. The failover



(a)



(b)

Figure 3.5: Performance of (a) Echo (upper line: with failure; lower line: without failure) and (b) Interactive (upper line: with failure; lower line: without failure).

time is the difference in the values of these two curves. Table 3.2 summarizes the failover time for the three applications. At a high HB frequency (i.e., at a short HB interval), the failover time is only a few hundred milliseconds, which makes it insignificant compared to the total time taken by the application in most cases. This is especially true of bulk

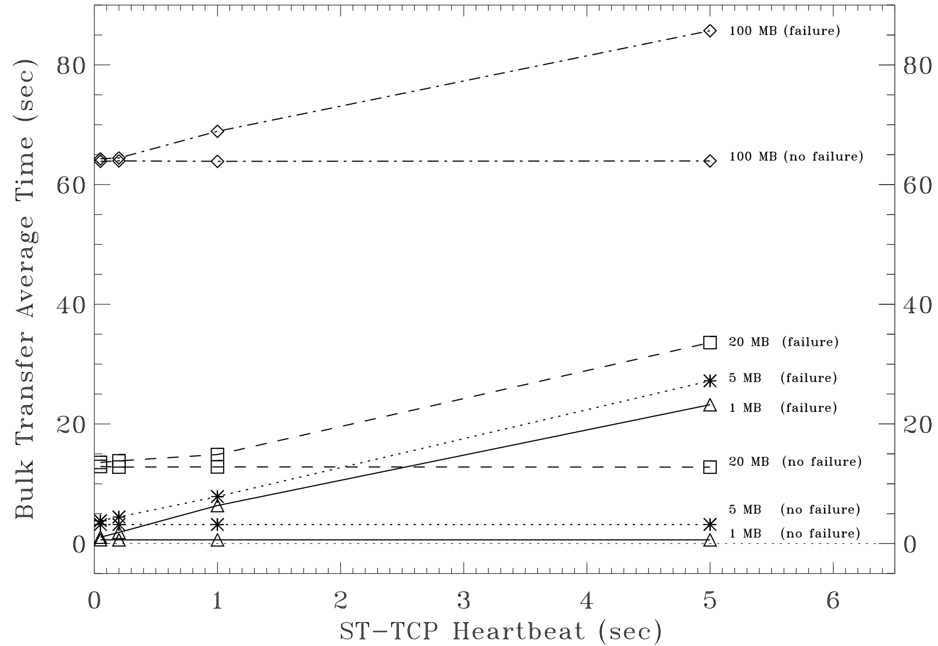


Figure 3.6: Bulk Transfer Application run with ST-TCP. Graph shows the total time taken with a failover and without failure (lower curve with same line type) during a run for data transfers of sizes 1 MB, 5MB, 20MB and 100MB.

transfer, Figure 3.6. Thus, by using an HB interval of about 50 milliseconds, ST-TCP ensures that there is no performance overhead during failure-free periods, and failover is quite short (less than 700 ms). This demonstrates that ST-TCP provides a fast failover.

3.6 Enhancements to ST-TCP

3.6.0.1 TCP state exchange between primary and backup

In the earlier architecture of ST-TCP [47], described in the previous sections, the backup received not only all traffic from the client to the primary, but also all traffic from the primary to the client. We observed that this leads to an overloaded NIC (network interface card) or/and CPU on the backup server. In particular, in some scenarios, the backup starts lagging behind the primary. In the initial ST-TCP prototype, the primary interpreted this situation as the backup being failed. This performance issue

	Failover Time (in secs)					
	Echo Application	Interactive Application	Bulk Transfer Application			
			1 MB	5 MB	20 MB	100 MB
ST-TCP 5s HB	22.309	23.832	22.580	24.013	20.805	21.764
ST-TCP 1s HB	5.524	6.187	5.723	4.667	2.067	5.022
ST-TCP 200ms HB	0.953	1.468	1.245	1.255	1.036	0.485
ST-TCP 50ms HB	0.219	0.412	0.417	0.627	0.676	0.422

Table 3.2: ST-TCP failover time for the three applications.

was addressed, to some extent, in the initial ST-TCP prototype by adding an additional NIC and CPU. One NIC could be used for the client-primary traffic while the other could be used for the primary-client traffic.

However, during our experiments, we noted that the backup does not need to receive the primary-client traffic at all. These segments were used by the backup for the following purposes:

(1) The backup examines the sequence numbers acknowledged by the primary in these segments to determine if it has missed any client segments that were received by the primary (and not by the backup).

(2) The backup uses these segments as an indication in some situations that the primary has crashed, e.g., if the primary app. crashes but the HB stays up. For example, consider the failure scenario where the application on the primary hangs and stops sending data to the client but the heartbeat between the primary and the backup stays up. If the backup is generating (and suppressing) segments for the client and observes that the primary is not, then the backup concludes that the primary has failed and takes over the TCP connection.

Both of these requirements can be addressed without the backup receiving primary-client segments. This can be achieved by having the following information included in the heartbeat messages exchanged between the servers - A) the sequence num-

ber of the latest byte received from the client (`LastByteReceived`), and B) the sequence number of the latest byte written to the TCP send buffer by the application (`LastAppByteWritten`). The backup can use the information in A and B for recognizing conditions mentioned in 1 and 2 above, respectively.

The current design of ST-TCP implements this new mechanism and so does not need any additional hardware. Also, this ensures that the backup does not receive and process any more traffic than the primary. However, it is required that the backup machine be preferably faster or at least as fast as the primary. Further, the workload on the backup should not be any more than that on the primary. This would ensure that during normal operation the backup does not excessively lag behind the primary, which could make the primary suspect that the backup has failed. Note that the primary lagging behind the backup is not an issue since the client sends and acks data depending on the primary's response.

3.6.1 Heartbeat Mechanism

In the earlier architecture of ST-TCP [47], the heartbeat (HB) mechanism was implemented by a UDP channel over the IP link. This mechanism created some scenarios where a single failure could not be correctly detected. For example, if the backup NIC failed, the backup would stop receiving regular heartbeat messages and conclude that the primary has failed. In this situation, it will shut down the primary and attempt to take over the TCP connection.

To address this problem, in the new ST-TCP architecture, the HB is exchanged between the primary and the backup over two diverse links. One is over the IP link as before; the second is over a serial link. This secondary link is established by directly connecting the serial ports of the two machines using a null-modem cable. These dual HB links allow the primary and secondary to continue to exchange HB information despite single failures. It also provides a better failure detection in some scenarios, e.g.,

the one outlined above.

Heartbeat carries the following information: (1) Last byte received from the client (`LastByteReceived`); (2) Last ack received from the client (`LastAckReceived`); (3) Last byte written by the application to the TCP send buffer (`LastAppByteWritten`); and (4) Last byte read by the application from the TCP receive buffer (`LastAppByteRead`). In addition, the information about the generation of a TCP FIN or TCP RST segment is also communicated through the HB. In situations where the primary and the backup send ping requests to the gateway (see Section 3.7.5), the results of these requests are exchanged via the HB.

The serial link uses RS-232 protocol and typically has a maximum speed of 115.2 kbps. The HB is less than 20 bytes per TCP connection, and assuming a HB every 200ms, this translates to a bandwidth of 0.8 kpbs per TCP connection. Thus, the serial link provides enough bandwidth for around 100 simultaneous TCP connections. If it is expected that the server will be supporting more connections, then it is best to use an additional 10/100 mbps Ethernet NIC on the primary and backup instead of a serial connection. The NICs can be directly connected by a crossover Ethernet cable.

3.6.2 Application Crash Failure

One limitation of providing server fault tolerance at the TCP layer is that it may not be able to adequately handle some server application failures. In particular, consider a situation where the application running on one of the servers (primary or backup) crashes, while its replica running on the other server continues to work correctly. Server fault tolerance provided at the TCP layer is limited in its ability to handle all possible scenarios that may arise under this situation.

In the new ST-TCP design, we have enhanced failure detection mechanism to detect most application crash failures. All application crash failures cannot be detected since ST-TCP is limited to the information available at the transport layer. Further,

since it is a primary-backup system, if the primary application and its replica differ in their response, e.g., one produces a FIN and the other does not, additional information is needed to determine whether the primary or the backup application has failed. The various application failure scenarios including the specific failure instances that ST-TCP may not be able to detect are described in the next section.

3.7 Failure Scenarios and Recovery Actions

An important goal of ST-TCP is to tolerate all single crash failures at the server. These failures could be at the hardware, operating system or application level. In the initial ST-TCP prototype, fault tolerance mechanisms mainly addressed HW/OS crashes. However, a common failure scenario is one in which there are no HW/OS crashes and one of the application replica (primary or backup) crashes or hangs, while the other continues to function correctly. This can happen because of differences in resources available on the primary and backup server, e.g., the application on the primary may run out of memory, while the application on the backup has sufficient memory available. We have enhanced ST-TCP to address these scenarios.

A crash failure model is assumed (Byzantine failures are not supported). ST-TCP handles all HW/OS failures, and all those application failures whose underlying cause can be traced back to the HW/OS (e.g., failures arising from memory allocation errors). Specifically, ST-TCP cannot handle failures resulting from software bugs in the application since these are likely to manifest both on the primary and the backup (and thus would count as a double failure). When a failure occurs at the primary server, the client sees one of the following three events: (1) No response from the server, e.g., a crash failure of the server due to HW or OS crash. (2) A TCP FIN or RST segment from the server indicating socket closure, e.g., OS closes socket after application has crashed. (3) A TCP RST segment from the server, e.g., the application on the primary crashes and a FIN is either not generated or does not reach the client; if the client now

sends some data (before the backup detects primary's failure and takes over the TCP connection), the primary would send back a RST since the socket does not exist any more.

In ST-TCP, single crash failures are masked from the client by suppressing the corresponding failure event and migrating the TCP connection to the backup. Table 3.3 summarizes all single crash failure conditions, symptoms observed and recovery actions taken.

	Failure	**Location**	**Symptom**	**Recovery Action Taken**
1	HW/OS crash failure	Primary	Backup detects HB failure on both links	Backup takes over the TCP connection and shuts primary down
		Backup	Primary detects HB failure on both links	Primary runs in non fault-tolerant mode and shuts backup down
2	Application failure (TCP FIN/RST not generated)	Primary	Primary app. lags backup app. by `AppMaxLagBytes` or by `AppMaxLagTime` (see Section 3.7.3)	Backup takes over TCP connection and shuts primary down
		Backup	Backup app. lags primary app. by `AppMaxLagBytes` or by `AppMaxLagTime` (see Section 3.7.3)	Primary runs in non fault-tolerant mode and shuts backup down
3	Application failure (TCP FIN/RST generated)	Primary	TCP FIN/RST generated at primary but not at backup; same symptoms of app. failure as in 2 above (see Section 3.7.4)	FIN/RST suppressed for `MaxDelayFIN`; if failure detected, backup takes over the TCP connection and shuts primary down
		Backup	TCP FIN/RST generated at backup but not at primary; same symptoms of app. failure as in 2 above (see Section 3.7.4)	FIN/RST discarded; if failure detected, primary runs in non fault-tolerant mode and shuts backup down
4	NIC or cable failure	Primary	Both primary and backup detect HB failure on IP link (but not on serial link); backup receives client data but primary does not, or, backup can ping gateway, but primary cannot (see Section 3.7.5)	Backup takes over the TCP connection and shuts primary down
		Backup	Both primary and backup detect HB failure on IP link (but not on serial link); primary receives client data or acks but backup does not (see Section 3.7.5)	Primary runs in non fault-tolerant mode and shuts backup down
5	Temporary Network failure	Backup	HB on both links up; backup does not receive some client bytes received by primary	Backup server requests and receives missed bytes from the primary
		Primary	Primary misses bytes; client retransmits	None required; normal TCP behavior

Table 3.3: Single Failure Scenarios

3.7.1 HW/OS Crash Failures

A HW/OS crash failure causes the primary to stop sending/receiving any data on the TCP connection (i.e. it does not send or ack any bytes). The backup concludes that the primary server has crashed if it detects HB failure on both links (IP and serial links). The underlying assumption here is that a single failure cannot take both links down, unless the primary has failed. In this case, the backup takes over the TCP connection and shuts the primary down.

Similarly, the primary concludes that the backup has crashed if it detects a HB failure on both links. The primary shuts the backup down and runs in non fault-tolerant mode.

3.7.2 Application Crash Failures

Managing application failures is more complicated. Since HW/OS is functioning correctly in these scenarios, the TCP layer stays up and HB between the servers also stays up on both the links. It is convenient to consider two separate cases here depending on whether the OS cleans up or does not clean up the failed application state.

3.7.3 Application Crash Failure without Cleanup

In this case, the application has failed, but the TCP socket is not closed. This can happen if neither the OS detects the application failure nor does the application itself close the socket due to the failure. The key point here is that a TCP FIN or RST segment is not generated. No data is sent or received by the application layer, but client bytes received by TCP are acknowledged by the TCP layer as long as the receive buffer does not fill up. Further, any bytes already in the send buffer may be sent out to the client. If the application running on the primary fails in this manner, the backup can observe one or both of the following about the primary application: (1) The application does not read any bytes from its TCP receive buffer (2) The application does not write

any bytes to its TCP send buffer.

The last byte read and written by the application on the primary is available to the backup through the HB mechanism. Failure detection is based on the following two parameters: (1) The number of bytes that the primary application lags the backup by (`AppMaxLagBytes`) (2) The time duration for which some number of bytes, already read or written by the backup application, have not been read or written by the primary application (`AppMaxLagTime`).

The two threshold values, `AppMaxLagBytes` and `AppMaxLagTime`, are configurable parameters and influence the failover time. A simple failure detection criteria would be: the primary application is considered failed if it lags behind the backup application by `AppMaxLagBytes` for a short duration of time (e.g., a few sec.) or, a particular byte read/written by the primary application lags the corresponding one at the backup by `AppMaxLagTime`. It is possible that the primary application has not really failed but just degraded in performance. However, if the failure criteria are met, the performance degradation is considered to be severe enough to warrant a failover. There is no danger of a dual active server here, because the backup powers down the primary [47] before taking over the TCP connection. The primary uses a similar criteria for detecting failure of the backup application.

In some instances - when there is no activity on the connection - failure detection may be delayed. However, these failures will be detected when the connection is used again. It should be noted that ST-TCP detects all application failures of the type discussed in this subsection, that is, where a FIN or RST segment is not generated.

3.7.4 Application Crash Failure with Cleanup

In this case, the application failure is detected by the OS. As part of the application cleanup, the OS closes the TCP connection. An example of such a failure is an application crashing as a result of receiving a SEGV signal. Such a failure could also

occur if the application detects a failure (e.g., a memory allocation error) and closes the TCP connection.

The main challenge in detecting this kind of failure is to be able to distinguish between a TCP FIN generated due to a normal closure of the socket and that generated due to an abnormal one. To understand the complexity of this scenario, consider that the application on the primary fails, and the TCP on primary generates a FIN as a result. If this FIN is sent to the client, the TCP connection will be terminated. This is despite the fact that the application on the backup is running correctly.

To address this scenario, ST-TCP requires that a server generating a FIN should immediately communicate the FIN to the other server through the HB. If the primary generates a FIN, it sends it to the client as soon as it learns (via HB) that the backup has also generated a FIN. This scenario is a normal closure of the TCP socket. While a failure can also result in both the primary and the backup producing a FIN, it is a case of double failure and is not currently handled by ST-TCP.

The interesting cases are where the primary and the backup disagree, i.e., only one of them produces a FIN. In this case the server generating the FIN delays sending it to the client for a short period of time, `MaxDelayFIN`, e.g., 1 minute. This is to account for cases where a failure may be detected via other indications during this time. In fact, by delaying the FIN temporarily, this failure scenario during the delayed time period becomes identical to the one described in Section 3.7.3, where no FIN is produced. However, if at the end of `MaxDelayFIN` a failure is not detected, it is assumed that the behavior of the primary is correct.

Suppose the TCP on the primary generates a FIN (either normal close or application failure). The primary will communicate to the backup via HB that it has generated a FIN, and delay sending it to the client. If the backup is functioning correctly and does not generate a FIN, it will detect the primary application's failure using the methods described in Section 3.7.3. As a result, it will shut off the primary and take over the

TCP connection before the delay time period (`MaxDelayFIN`) is over. On the other hand, suppose that the primary application failure is not detected by the backup. It is highly likely that the FIN on the primary was generated due to normal operation and it is the backup that has actually failed and has not produced a FIN. In this case, the primary sends out the delayed FIN to the client, shuts down the backup and runs in non fault tolerant mode. We decided not to do a failover whenever the primary produced a FIN and the backup did not, since it is quite possible that it is the backup that has failed and hence not produced a FIN for normal socket closure. It should be noted that the primary always immediately sends out a FIN if it has already received a FIN from the client. Furthermore, note that during normal operation – when neither the primary nor the backup has failed – the FIN is not delayed by `MaxDelayFIN`. This only happens if there is a failure.

All failure scenarios where only one of primary or backup generates a FIN are summarized below.

(1) Primary application has failed; Backup is working correctly.

Primary generates a FIN, but backup does not generate a FIN. Here the primary application fails and a FIN is generated. The primary delays sending the FIN for `MaxDelayFIN` time units. During this time it is likely that the backup will detect the primary failure (if the application performs read/write operations on the socket). The backup will shut the primary down and take over the TCP connection. However, if the failure is not detected within `MaxDelayFIN` time units, the primary will send out the FIN to the client.

Primary does not generate a FIN, but backup generates a FIN. Here the primary application fails and a FIN is not produced. The backup generates a FIN due to normal socket closure. This FIN on the backup is dropped like any other segment sent to the client. The FIN is treated specially here, and

although it has a sequence number, it is not considered in the failure detection criteria. The backup will detect the primary application failure if there are other bytes that the backup application reads/writes but the primary does not. In that case, it will shut off the primary, take over the TCP connection, and retransmit the FIN (in fact, the backup has already been retransmitting and dropping the FIN).

(2) Primary is working correctly; Backup application has failed.

Primary generates a FIN, but backup does not generate a FIN. In this case, the primary produces a FIN due to normal socket closure, but since the backup application has failed, the backup does not generate a FIN. Here the primary will wait for at most `MaxDelayFIN` time units before sending out the FIN. Meanwhile, if it detects backup's application failure, it will send out the FIN immediately.

Primary does not generate a FIN, but backup generates a FIN. Here the backup TCP generates a FIN due to a failure. This FIN is suppressed. The primary transits to non fault-tolerant mode. This happens at `MaxDelayFIN` if the primary is unable to detect backup's application failure; otherwise, it happens at the time the primary detects backup's application failure.

In the application failure cases described above, the failures are very likely to be detected if the application is reading/writing bytes when the failure occurs. However, if there is no application activity, the application failure may not be immediately detected. This is not a problem for application failures described in the previous section where a FIN is not generated (since failure will be detected when there is some application activity). However, in application failure situations where a FIN is generated, it may not be possible to conclusively distinguish between a normal closure and a failure based solely on information at the TCP layer. This is true for any primary/backup system.

To be able to detect application failures under all circumstances, either additional backup servers have to be deployed, or some additional information is needed from the application layer. Deploying additional backup servers will allow a majority decision to be taken in case of a conflict between the primary and a backup. For additional information from the application layer, an application can support a watchdog mechanism where the application continually sends a heartbeat to a watchdog. The watchdog monitors the application health and informs ST-TCP in case of any failure suspicion.

A TCP RST segment is usually generated for the client when the client sends a segment on a TCP socket that no longer exists on the server. However, the socket API has a `SO_LINGER` socket option that can modify the behavior of TCP when a socket is closed. This option allows an application to specify a timeout value. If all the bytes queued on the connection have not been sent reliably within this timeout value after a `close()` is called on the socket, TCP terminates the connection by sending a RST to the far end. Although not recommended, a timeout value of zero is also valid, in which case, TCP will always terminate that connection with a RST when the socket is closed. Since, like the FIN segment, the RST segment can also be generated both in failure and failure-free scenarios, we treat RST segments produced at the primary and the backup very similar to the FIN segments.

3.7.5 Local Network Failures

In this section we discuss network failures that are local to the primary or the backup server, e.g., a NIC failure in a server. We have assumed that both the primary and the backup have a single NIC. Recall from Section 3.6 that the HB between the primary and the backup is exchanged on two separate links – an IP link and a serial link.

If a local network failure occurs, only the HB on the IP link fails. The servers continue to exchange the HB on the serial link. This enables the two servers to determine

that a local network failure has occurred. To determine if the failure has occurred at the primary or at the backup, the servers examine the "last client byte received" information (`LastByteReceived`) in the HB. If the client is sending data, then the server with the NIC failure will not receive them, while the server without NIC failure will receive them. Based on the `LastByteReceived` in the HB, the primary or the backup can determine if the other is lagging behind in terms of the client bytes received. For example, if the backup determines that the primary has lagged behind by greater than a threshold number of bytes, or, that a particular byte has not been received by the primary for more than a threshold period of time, then it shuts the primary down and takes over the connection. These threshold values are configurable. Similarly, the primary shuts the backup down if it determines that the backup is lagging behind.

One limitation of this failure detection method is that it depends on the client sending data. There are several applications, e.g. FTP, that do not require client to send a lot of data. In such cases, this method of failure detection does not work. This problem can be partially solved by having the primary and backup look at the acks received from the client. If the backup NIC is down, the latest client ack information (`LastAckReceived`) received by the primary from the backup via the HB on the serial link will indicate that the backup is behind. However, this does not work if the primary NIC has failed. If the primary NIC is down, the client will not receive any bytes from the server and thus not send any acks.

We have added another mechanism in the new version of ST-TCP to address this case. When the servers detect a failure of the HB on the IP link but not on the serial link, both the primary and the backup send ping requests to their gateway. The results of these requests - that is, if they succeeded or not - are exchanged in the HB via the serial link. If ping requests continue to fail for the primary but succeed for the backup, the backup takes over the TCP connection and shuts the primary down.

Temporary local network failures. Temporary failures in the NIC or the IP

stack (e.g. buffer overflow) can lead to packets being dropped. HB stays up on both the links in this case. If the packets are dropped at the backup, the backup requests the primary for the missing bytes. There may be cases where the backup takes a long time to catch up or is unable to catch up. If the additional receive buffer space at the primary fills up, the primary considers the backup failed and runs in non fault-tolerant mode. Note that temporary network failures at the primary are not an issue since these will be taken care of in the normal course of TCP operation – the primary does not ack these bytes and therefore the client will retransmit.

If the primary crashes while the backup is retrieving missed bytes from it, the backup has no way of obtaining these bytes, since primary has already acked them. For critical applications, a logger can be added to the system to address this output commit problem as described in [47]; for other applications, ST-TCP treats this failure as unrecoverable.

3.8 Summary

In this section, we described the design, implementation, and performance evaluation of ST-TCP, which is an extension of TCP to tolerate TCP server failures. This is done by using an active backup server that taps the TCP bytes exchanged between a client and primary TCP server, maintains a consistent state of the TCP connection, and takes over the TCP connection whenever the primary server fails. This migration of the TCP connection to the backup server is completely transparent to the client. Because no changes are required on the client machine, any TCP client can access any ST-TCP server.

ST-TCP has been implemented in Linux operating system by making a few changes in TCP/IP stack in the kernel. The modified version runs on primary and backup server machines. A performance measurement from this prototype implementation shows that the performance overhead of ST-TCP over standard TCP is insignificant

when there are no failures. In addition, the failover time exhibited by ST-TCP is quite low, less that 700 milliseconds when the primary and backup servers exchange heartbeat messages every 50 milliseconds.

The key distinguishing features of ST-TCP are complete transparency to the clients, no performance overhead during failure-free periods, and a fast failover that clients are unlikely to notice, especially in long-running applications. The design and prototype implementation of ST-TCP demonstrates that ST-TCP is a better alternative to tolerate TCP server failures than other approaches proposed earlier.

Chapter 4

Fault-Tolerant and Scalable TCP splice

This Chapter describes three enhancements to the TCP splicing mechanism: (1) Enable a TCP connection to be simultaneously spliced through multiple machines for higher scalability; (2) Make a spliced connection fault-tolerant to proxy failures; and (3) Provide flexibility of splitting a TCP splice between a proxy and a backend server for further increasing the scalability of a web server system. A prototype of these enhancements have been implemented as a Linux 2.6 kernel module, and important performance results measured from this implementation are presented.

4.1 Introduction

Internet services are commonly offered over the web using application layer proxies. These proxies perform a number of important functions, including layer-7 or content-based routing that routes different client requests to the appropriate application servers based on request content. Other functions performed by these proxies include web content caching, implementation of security policies (e.g., authentication, access control lists), implementation of network management policies (e.g., traffic shaping) and usage accounting.

TCP splicing [44, 70] has been commonly used for improving the performance of serving web content through proxies. It avoids any context switches or data copying between kernel and user space, resulting in improved performance. In fact, it has been

shown that TCP splicing makes the performance of a proxy comparable to that of IP forwarding [43]. Advantages of using TCP splicing to build web servers are described in [62, 61, 22].

At present, a TCP splicing based web server architecture suffers from two drawbacks: (1) All traffic between clients and servers (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, clients have to re-establish their HTTP connections and re-issue failed requests, even in the presence of a backup proxy.

This work addresses these two drawbacks of a TCP splicing based web server architecture. In particular, it consists of providing three enhancements to the TCP splicing method. These are: (1) design and implementation of a replicated TCP splice; (2) design and implementation of a fault-tolerant TCP splice; and (3) design and implementation of a split splice. These enhancements allow a TCP connection to be spliced at multiple proxies, providing both fault-tolerance and higher scalability. In particular, failure of a proxy causes no disruption to a splice, and additional proxies can be added to scale the system. To further increase the scalability of a web server architecture, a TCP splice can be split between the proxies and a backend server, with the backend server performing the splicing functionality for the response packets. We describe the design, implementation and evaluation of these three enhancements to TCP splice.

A large body of research exists on architecture of web server clusters, focusing mainly on scalability and efficiency [8, 9, 32, 39]. Fault-tolerance is usually assumed to come for free by virtue of using a cluster and replication of data. In the event of a server failure, the client is expected to retry its request. Although this works well for traditional web service applications that mainly download webpages to the client machines, it is unacceptable for newly emerging web service applications that are long-duration, highly interactive, and stateful in nature.

In the next Chapter 5, we describe the design, implementation and evaluation of a

web server architecture that addresses this limitation. This new architecture is based on the three enhancements to the TCP splicing functionality, and provides enhanced fault-tolerance. It supports content-based request distribution (layer 7 routing), including support for HTTP 1.1 persistent connections. The enhanced fault-tolerance results in minimal user impact in the event of server failures, with outages not lasting more than a few seconds. It is provided without sacrificing scalability or efficiency. In fact, the new architecture is as scalable and efficient as the earlier web server architectures [9, 39] that do not provide such enhanced fault tolerance support.

An important consideration of our work is to realize a server architecture with inexpensive, off-the-shelf components. Although, use of specialized hardware, e.g., a network processor, can improve performance [81], it is significantly more expensive than a network interface card (NIC). Furthermore, our architecture could be implemented in a network processor leading to a better performance.

The rest of the chapter is organized as follows. In the next section, we provide a brief description of TCP splice and discuss some related work. In Section 4.3, we describe the overall system architecture. This is followed by a description of some architectural configurations. Load balancing and proxy architectures are presented in Sections 4.5 and 4.6, respectively. This is followed by sections on our prototype implementation of the system on Linux and experimental results. Finally, a summary is presented in Section 4.9.

4.2 Background

4.2.1 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, and, the associated context switches. TCP Splice was proposed [44, 70] 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, which is done entirely in the kernel. This makes the latency and computational cost at a proxy only a few times more expensive than 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 [62, 61, 22].

The following steps are required in establishing a TCP splice.

- The client connects to a proxy. The proxy accepts the connection and receives the client request.
- It may perform authentication and other functions as configured by the administrator. Then it performs layer 7 routing to select a server and initiates a new TCP connection to it.
- At this point, 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 modification of the header (which 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 on to the client (after header modifications).

4.2.2 Related Work

The design of server architectures has been an active area of research for the past several years. The main focus of this research has been on enhancing the typical

server architecture to make it highly scalable, responsive, reliable and cost effective. A good survey of some of the earlier web server architectures and the related issues is given in [18]. Content-based routing using a layer-7 switch allows the front-end to parse in-coming requests and make a decision about which backend server to dispatch the request to [21, 3]. Research has shown that content-based routing significantly improves the scalability of web servers. TCP splicing[43, 44] and TCP handoff [56] mechanisms were introduced to optimize content-based routing approach.

Based on where the functionalities of receiving client requests, parsing client requests, and forwarding client requests to the appropriate backend server (content-based routing) are implemented, we can classify current server architectures into three categories: front-end based systems, backend based systems and hybrid systems. Advantages of front-end based systems include (i) cluster management policies and administration are encapsulated in a single machine; and (ii) backend servers do not require any changes and hence can be unaware of the cluster. The disadvantages are (i) low scalability, since front-end is a bottle-neck; and (ii) front-end is not fault-tolerant. Examples of front-end based systems are [21, 36].

Advantages of backend based systems include high scalability and server fault-tolerance. The main disadvantage is that they require modifications to the OS of the backend servers. Examples of backend based systems are [39, 9, 74]. Finally, the advantages of hybrid systems are high scalability, server fault-tolerance, and an ability to efficiently manage subsequent requests from the same clients. The main disadvantage is that they require modifications to the OS of the backend servers. Examples of hybrid systems are [32, 8].

4.3 Architecture overview

Figure 4.1 shows the functional components of our architecture, which consists of two main stages – a proxy stage and a backend server stage. A stateless load balancer

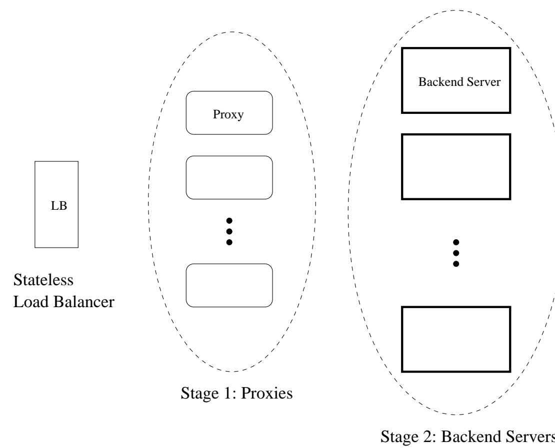


Figure 4.1: Functional components of our architecture.

(LB) precedes the proxy stage. This LB distributes the packets received among a cluster of proxy machines. Following the proxies is the backend server stage where the client requests are processed and responses generated. Note that this is only a functional representation of our architecture and that in an actual implementation some of these components may be co-located, e.g., the LB could be co-resident with a proxy, or, a proxy may be co-resident with a backend server.

The LB works at the IP layer, is stateless and uses a simple load balancing algorithm to evenly spread the load among the proxy machines. It spreads the load at the IP level, and does not modify the client packets in any way. The fact that the LB is completely stateless is a significant advantage of this design, since it makes failure recovery quick and trivial. A primary-backup fault-tolerance scheme can be used with virtually no need for synchronization, because of the stateless nature of the LB. Multiple LBs can be used for fault-tolerance, however, as shown in [9], a single LB can handle a very high packet rate and is usually not a bottleneck.

The proxy stage first accepts a client's TCP connection and HTTP request, and then performs an L7 or content-aware routing and load balancing to pick a backend server suitable for handling that request. Content-aware routing has been extensively

studied [18, 19], and its main advantages are: (1) Convenient partitioning of web server content among the servers leading to storage efficiency and scalability; (2) Consolidation of specialized content on specific servers, e.g., one set of servers for video content, and another set of servers for audio; and (3) Better performance by exploiting cache affinity [56].

It should be noted that our architecture supports both systems, one where a separate physical proxy is required and the other where it is not. A separate pool of proxy machines can be used if such is required for other reasons such as security. Otherwise, a machine can serve both as a proxy and a server. Although, in the rest of the chapter, we talk about proxy machines and backend servers, it should be noted that this distinction is made purely on a functional basis. Indeed, a proxy and a backend server may be implemented on the same physical machine. An advantage of not dividing up the machines into proxies and backend servers is that then we do not need to determine the ratio in which they should be divided, which is typically dynamic and not trivial to arrive at. An incorrect partitioning of the servers could lead to the proxy machines or the backend servers becoming a bottleneck.

The proxy, after choosing a backend server, splices the client TCP connection with the connection to the backend server. There are two issues we need to address at this point: (1) The proxy that creates the TCP splice is a single point of failure; and (2) All traffic on the spliced connection (both directions) must pass through the proxy that created the splice, potentially making it a performance bottleneck.

Replicated Splice: To address these issues, we replicate the state information required to perform the splicing to all other proxies. This enables any proxy to splice any already spliced connection. In other words, a particular spliced connection is not bound to any particular proxy. This design results in three key advantages: (1) The TCP splice is distributed, that is, subsequent segments of the same spliced connection can pass through different proxies. (2) The TCP splice is fault-tolerant to the failure

of a particular proxy machine. In fact, if a proxy machine fails, no explicit recovery action needs to be taken, other than the detection of the proxy failure by the LB, so that future packets are not sent to the failed proxy. (3) By replicating a TCP spliced connection, the server response can now pass through any of the proxy machines, thus eliminating/reducing the above-mentioned performance bottleneck.

Split Splice: In the backend server stage, client requests are processed and appropriate responses are generated. The response is sent to a proxy where it is spliced and sent to the client. An advantage of using TCP splicing is that no changes at all are required at the backend servers, which could be running any OS. However, having to send the response packets through a proxy does limit the scalability of the system. As mentioned above, replicating a TCP spliced connection does eliminate/reduce the above-mentioned performance bottleneck. For enhanced scalability, we modify the splicing functionality so that the architecture has the flexibility of allowing a backend server to directly send a response to a client without it having to pass through a proxy.

A TCP splice can be viewed as a combination of two uni-directional splices: a client-to-server splice and a server-to-client splice. The client-to-server splice handles header manipulation of TCP segments from a client to a server, while the server-to-client splice handles header manipulation of TCP segments from a server to a client. At present, both of these unidirectional splices are implemented as a single module on the same machine. We relax this requirement of co-locating the two uni-directional splices on the same machine. In particular, we implement the client-to-server splice in the proxy and the server-to-client splice on the server. Thus, while the proxies splice TCP segments from the client to the backend server, the backend server itself performs the splice of the response segments and sends them directly to the client.

This idea of splitting a splice into two half-splices is similar in spirit to the splitting of a TCP connection into two pipes as proposed in [39]. However, unlike [39], our proposal does not require any complex changes in the existing TCP/IP stack, nor does it

require a new protocol like split-stack in [39] to synchronize the two splices. Splitting the splice functionality does require some changes to the OS of a backend server. However, in cases where such changes are not possible for some backend servers, the architecture supports mixed configurations, that is, some backend servers support a split splice while others do not.

Although the first request on a client connection is L7-routed by a proxy to a backend server, subsequent ones on the same connection are routed by a backend server. If a new backend server is more appropriate of the request, the proxies are informed so that they can "resplice" the connection to the new backend server.

4.4 Flexible Server Configurations

A key guiding principle of our proposed architecture is that the server components are inexpensive and available off-the-shelf. In fact, our proposed architecture provides a flexibility of merging the proxy and backend server functionalities on the same set of machines, not changing the OS of backend servers at all, or even a mixed configuration where the OS of some backend servers is modified, while no changes are made to the OS of other backend servers. Figure 4.2 illustrates three server configurations.

In the first configuration, the proxy functionality is implemented on dedicated machines, and no changes are required to OS on backend servers. This architecture is suitable for organizations that require separate proxy machines for reasons, such as, security, and the web service application requires bulk data transfer from client to server, e.g., a repository server for user videos and photos (similar to Flickr [27]). Replication of proxies (and hence the TCP splice) in this configuration results in high throughput, as the data from clients to servers can pass through separate proxies concurrently. Also, a proxy failure is transparently tolerated, since all subsequent data is forwarded via other proxies. Our experimental results, described in Section 4.8, confirm both of these features.

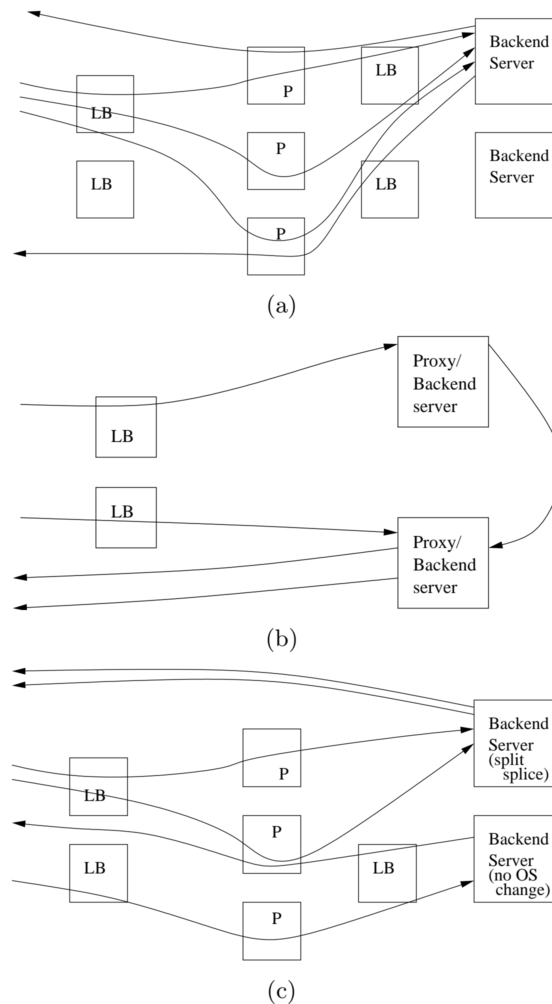


Figure 4.2: Some examples of possible server configurations. (a) Separate proxy machines; no backend server changes required. The figure shows the paths of packets belonging to a single connection. Packets in both directions need to pass through a proxy, but it could be any proxy. (b) Proxy and backend server on the same machines. A single connection is shown. Here the response packets are always directly sent to the clients. (c) Mixed configuration with separate proxies, and, split splice installed on some backend servers. The figure shows two connections, one on each backend server.

In the second configuration, the proxy functionality and backend server functionality are implemented on the same set of machines, potentially saving HW resources for the organization. OS changes are required on these machines. This configuration is suitable for organizations that do not require separate proxy machines, and, for web service applications that perform bulk data transfer to a client.

Finally, the third configuration is a mixed configuration. There are a small number of dedicated proxy machines, as required by an organization for security, and the OS of only some of the backend servers is changed. The OS of backend servers providing traditional web service applications is changed (split splice) to have the bulk data flow directly to the clients, bypassing the proxies. On the other hand, the OS of the backend servers that do not send bulk data to the clients remains unchanged.

Replication of TCP splice enables different requests or data from client to be routed via different proxies to a server. Failure of a proxy is tolerated by having subsequent packets routed via other proxies. However, response data from the servers to the clients still passes through one proxy for each connection. This obviously makes the proxy a failure bottleneck. This problem is addressed by using the split splice enhancement that allows a server to bypass proxies while sending a response. Split splice however requires modifications to a server. In situations where server modification is not feasible, a stateless LB, identical to the one used prior to the proxy stage, can be employed.

4.5 Load Balancing

Load balancing aims to uniformly distribute client requests among a group of servers, each of which are equally suitable for handling the requests. Ideally, requests should always be sent to the least loaded of the eligible servers. In our architecture, there are two instances where load balancing is required: (1) when incoming client packets are distributed among the proxies, and, (2) when a proxy selects a backend server for handling a client request. For (1), there is a choice of using a L2 load balancing technique like channel bonding, or, to use a L3 load balancer which can exist as a separate box, or be co-located with one or more routers or proxies. In either case, however, the packets are not modified in any way. For (2), a proxy acts as a L7 load-balancer.

4.5.1 Load Balancing Using Channel Bonding

An L2 switch that implements channel bonding can be used to implement load balancing. Channel bonding (also known as trunking or link aggregation or 802.3ad) can be used to load balance at line speeds. Conceptually, channel bonding allows a group of links/interfaces to be treated as one virtual entity. Two major plus points of channel bonding are load balancing and reliability. It load-balances among the links using a user specified algorithm. and also provides inherent, automatic redundancy on point-to-point links. So, should one of the multiple ports used in a link fail, network traffic is dynamically redirected to flow across the remaining good ports in the link. This redirection is fast and the network continues to operate with virtually no interruption in service.

In order to load balance client packet to the proxies, all the ports on an L2 switch connected to the proxies are bonded, as shown in Figure 4.3.

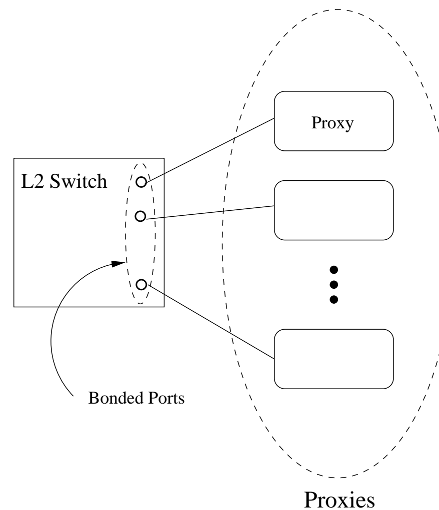


Figure 4.3: Channel bonding used for ports on an L2 switch.

4.5.2 L3 Load Balancer

There is an extensive body of research on L3/L4 load balancing. A survey of load balancing techniques, used in web server architectures, is provided in [18, 19]. In our prototype implementation, we have used this technique instead of channel bonding, because we did not have access to an L2 switch implementing channel bonding. We use a technique similar to that used in IBM's NetDispatcher [36], with a key difference – our LB does not keep any connection state information. In NetDispatcher and most other similar systems, the LB selects a server for a new TCP connection and maintains a mapping of the connection to that server. All subsequent packets belonging to that connection are sent to the same server.

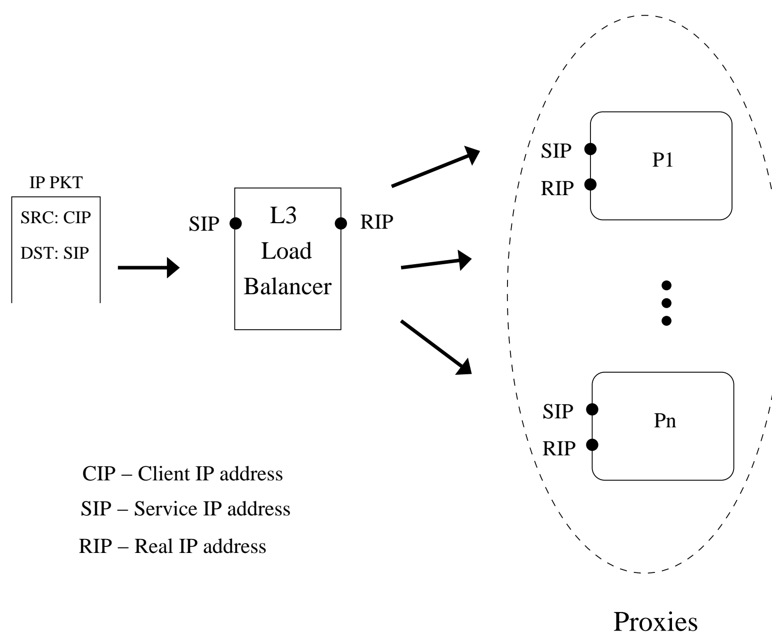


Figure 4.4: A layer 3 load balancer.

Our load balancer is shown in Figure 4.4. It is assigned a service IP address, which is the address that is advertised to the clients. All the proxies are also assigned the service IP address. A heart-beat (HB) mechanism exists between the LB and each of the proxy machines. This mechanism serves two purposes: (1) it allows the LB to

know if a particular proxy has failed, and, if that is the case, to stop sending packets to it; (2) as part of the HB, a proxy sends a "workload" factor which reflects the resources currently available on that proxy.

The LB uses a load balancing algorithm (described later) to determine which proxy to send the next packet to. Note that since the proxies also have the service IP address as a virtual address, no changes are required to the IP packet header; the packet is simply sent to the selected proxy by using the proxy's MAC address. Using the above mechanism for load balancing requires that the LB and all the proxies are in the same subnet. This requirement can be relaxed by the use of IP tunneling (although that may result in some additional overhead).

The LB is not a single point of failure as a backup LB can be used. The important point to note here is that the LB does not have any hard state; therefore no state information needs to be exchanged with a backup and failover to a backup in the event of a failure is simple and fast.

Our load-balancing algorithm is based on [11]. Consider a load balancer, L, and N servers. In order to balance load among the N servers, L maintains a heart-beat (HB) mechanism with each of the servers. L receives a workload factor, W_i, from the ith server every HB interval. These workload factors represent the amount of available capacity at a server and are determined based on factors, such as, number of existing connections, CPU and memory usage. Note that these factors are actually fractions of the maximum capacity, and, therefore, the actual capacity is not relevant and can be different across servers. The algorithm tries to schedule requests such that each server receives requests proportional to its workload factor. A convenient way to do this is by normalizing the W_i's: $W'_i = \frac{W_i}{\sum_1^N W_i}$. Then, to schedule a request, a random number, r, is generated and the request is scheduled to server, s_i, if $r \in [\sum_1^{i-1} W'_j, \sum_1^i W'_j]$.

[11] assumes that all the client requests have equal "cost", that is, would require same computing resources. This algorithm can be enhanced by weakening this

assumption and estimating the cost of each client request.

4.6 Proxy Architecture

Distributing and replicating TCP splice state information among all proxies allows packets belonging to a connection to flow through any of the proxies once the splice is established. However, to create a new TCP splice (for a new client-server session), the first client request, that is used to do layer-7 routing, must be processed by the same proxy that accepted the initial TCP connection. Notice that a TCP splice is created only after receiving this first client request (which may be several TCP segments long).

To ensure that before a connection is spliced, all segments are received by the same proxy, and, to distribute the load of creating spliced connections evenly, the set of proxy machines are divided into multiple groups. A new client connection is identified by a hash computed on source IP address, destination IP address, source port number and destination port number. We provide a mapping between this hash and a proxy group that creates the corresponding spliced connection. Figure 4.5 shows the sequence of steps involved in handling a client request. These steps are described below:

(1) The LB receives a client packet and uses its load balancing algorithm to forward it to a proxy.

(2) The proxy receiving the TCP segment determines if it is a new client connection. It does this by looking at the IP addresses and port numbers in the segment and comparing them to its TCP splice state information. If the segment matches an existing splice, it is spliced and sent off to the corresponding backend server. Otherwise, a hash is computed for it, and the proxy group that is assigned for creating its splices is determined. The segment is then multicast to the members of that proxy group.

(3) The proxies in the proxy group receive the segments. Here another hash is

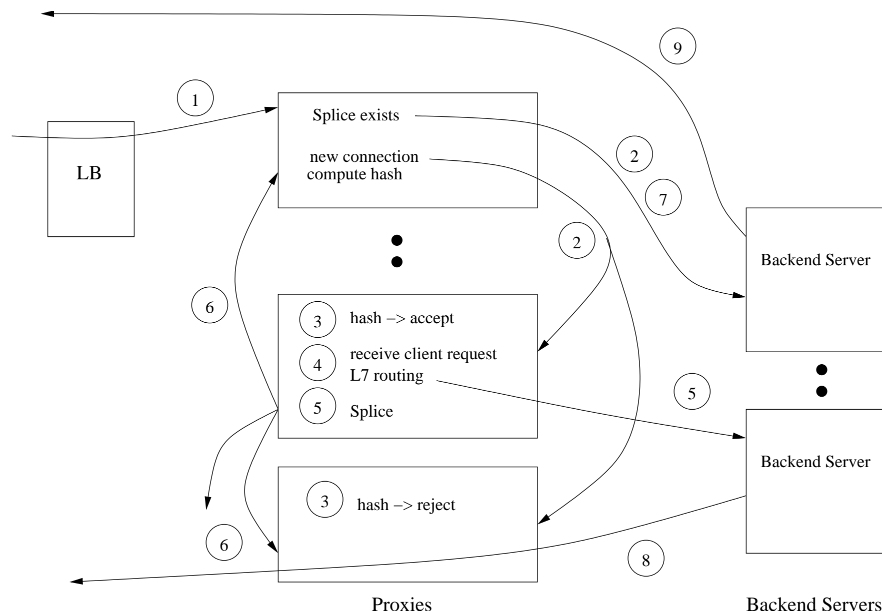


Figure 4.5: Sequence of steps involved in handling a client request.

computed and precisely one member of the proxy group accepts the segment.

(4) That proxy accepts the client TCP connection and waits for the client request to arrive. Once it has received the complete client request, it uses its L7 routing algorithm to find an appropriate backend server.

(5) The proxy opens a TCP connection with the chosen backend server, sends it the client request and splices the two TCP connections.

(6) The proxy then sends the splicing state information of this connection to all other proxies.

(7) Further segments from the client arriving on that TCP connection can now be spliced by any proxy to the backend server.

(8) The backend server on receiving the request, processes it and sends the response to the client. The server can use any proxy for the response.

(9) If split splice is installed on a backend server, the response is directly sent to the client without passing through a proxy.

4.7 Implementation

We have implemented a prototype of our system in Linux kernel version 2.6.12. This implementation consists of the following major parts: (1) TCP splicing code, including support for distributed splicing and split splicing; (2) Load balancing code; and (3) Proxy code.

In our implementation, we have used Netfilter [53] quite extensively. 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. Five such hooks are provided, as shown in Figure 4.6. These hooks intercept packets and invoke any CB functions that may be registered with that hook. When multiple CB functions are registered at a particular hook, they are invoked in the order of the priority specified at the time of their registration. 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.

4.7.1 Load Balancer

A software based load-balancer (LB) is used for the prototype. For better performance, a HW based load balancer could be used. However, since we are more interested in comparing performance rather than measuring the absolute performance of the system, a SW based LB is equally suitable for our experiments. Our LB, which is a Linux kernel module, is based on code from the Linux Virtual Server (LVS) project [42]. However, one key difference and advantage of our LB is that it is stateless, whereas LVS keeps a hash table of connections so that all packets of a connection are sent to the same server. The NF_IP_LOCAL_IN netfilter hook is used for intercepting packets. In the

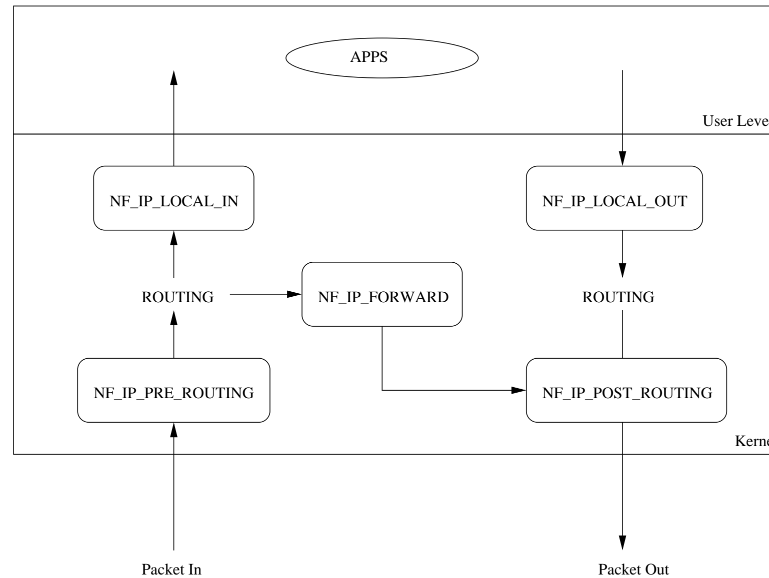


Figure 4.6: Netfilter allows functions to be registered at five different hooks in the network stack.

network stack, this is right after the IP packets have been defragmented. (This has the added advantage that the proxies do not have to worry about fragmented IP packets.)

The packets received for the service IP address are load-balanced among the MAC addresses of the proxies. Each of the proxy has a service IP address and a “real” IP address. These real IP addresses are configured at the load balancer. Using the load-balancing algorithm, the LB chooses a real server to which to deliver a packet, and then sends it out. The difference here from regular routing is that the packet is sent to the MAC of the proxy’s “real” IP address instead of the packet’s destination address.

For the experiments that follow, the load balancing algorithm used by the LB is round-robin. It sends 40 packets to a particular proxy before switching to the next one.

4.7.2 Proxy

The modifications made to the proxies can be divided into kernel and user level implementation.

Kernel level implementation. The kernel level changes at the proxies are implemented as Linux kernel modules called `tcpdistsplice` and `packetredirector`.

(1) `tcpdistsplice` This module provides the TCP splicing and distributed splicing functionality. It is based on the TCP splicing module available at the Linux Virtual Server (LVS) project [77]. It registers with the NF_IP_LOCAL_IN netfilter hook to intercept any arriving IP packets.

Netfilter also allows socket options and handlers to be defined. These are used for communication between a user process and `tcpdistsplice`. `tcpdistsplice` defines three socket options — `PRE_SPLICE, SPLICE, DIST_SPLICE`. In order to establish a splice between two TCP connections, a user process needs to call `setsockopt()` with `PRE_SPLICE`, and then with `SPLICE`. After receiving a `PRE_SPLICE`, the module drops and does not acknowledge any further packets from the client until the splice is established. The module also assumes that the server does not send any data after connection establishment until the client request is received. These restrictions can be relaxed, and, the splicing module made more efficient [70], however, that is not a focus of this work.

`DIST_SPLICE` is used for splicing on a remote machine. These splices are created based on state information, without the presence of the corresponding TCP level sockets. The `setsockopt()` call with `PRE_SPLICE` returns state information such as TCP sequence numbers related to the splice. This information is then communicated to other proxies to use with `DIST_SPLICE`.

(2) `packetredirector` This module ensures that all the client packets from a particular connection are handled by the same proxy before a TCP splice for that connection is created. It also uses the NF_IP_LOCAL_IN netfilter hook, but with a lower priority than the TCP splicing module, that is, `tcpdistsplice` is

always called first on a packet. If there is no match, that is, there is no splice for that connection, `tcpdistsplice` re-injects the packet back into the stack and `packetredirector` is called on it.

This module redirects packets based on computation of a hash. The hash is based on the source and destination IP addresses and port numbers. Note that packets that the LB sends to the proxies are already defragmented. After a TCP splice is created for a particular connection, and distributed to other proxies, packets belonging to that connection find a match in the splicing module and are never sent to the redirection module.

User level implementation. A user level proxy was implemented that — accepts a client connection, reads its request, performs L7-routing to connect to a server, passes splicing state information to other proxies, and then, finally, splices the client and server TCP connections. The significant parts of the proxy implementation are shown in Figure 4.7.

```
1 c1 = accept() // client connection
2 n = read(c1) // client request
3 s = L7routing() // determine server
4 c2 = connect(s) // to server

// do pre-splice
5 setsockopt(..PRE_SPLICE,c1,c2,info..)

// read any additional client data
6 n += read(c1) // (non-blocking)

//send splice info to other proxies
7 sendProxies(info, n)

// set up splice
8 setsockopt(..SPLICE,c1,c2,n..)

9 write(c2) // n bytes to server
```

Figure 4.7: Pseudocode of the proxy implementation.

A process called `spliceListener` also runs on a proxy. This process waits to receive splicing state information from other proxies, and then uses that information to

create a DIST_SPLICE.

4.7.3 Backend Server

There is an optional modification that can be made to backend servers. For greater scalability, the splicing functionality can be split between the proxies and the backend servers. We modify the `tcpdistsplice` module and register it at the NF_IP_LOCAL_OUT netfilter hook to implement this functionality. This hook is invoked right after a local process has sent out an IP packet. Other than the fact that it is registered at a different hook, this module is identical to `tcpdistsplice`.

The mechanism of sending the splice state information to a backend server and that of establishing a split splice there is identical to that of establishing a distributed splice at a proxy. Indeed, the functionality of a split splice at a backend server is identical to that of a proxy performing a distributed splice with the only difference being that the server split splice module will always receive packets in only one direction.

4.8 Experimental results

The goal of our experiments is to demonstrate fault-tolerance and scalability of our architecture. Our objective is not to test the capacity of a server. In particular, we perform three sets of experiments with a focus on (1) proxy scalability, (2) proxy failover, and, (3) split splice architecture.

4.8.1 Experimental Setup

We use seven machines for our experiments. Specific details of the machines are listed in Table 4.1. The experimental setup is shown in Figure 4.8. The machines are connected together using an 8 port, 1 Gbps D-Link Ethernet Switch. Although we use a separate machine as a load balancer, it is not necessary to do so. In our experiments the LB was not a bottleneck, confirming the results in [9]. In larger systems, as mentioned

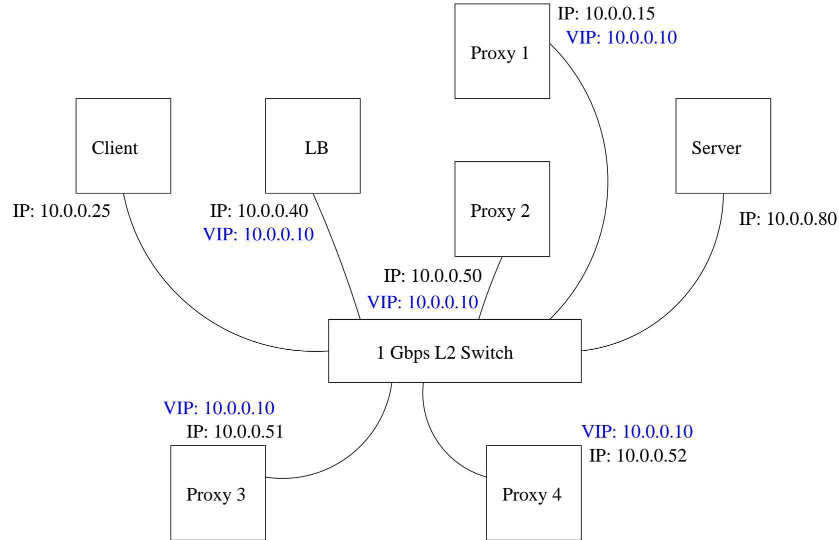


Figure 4.8: Experimental setup.

in Section 4.5, the load balancing function could be integrated with the routers or the proxies.

Client:	Shuttle xpc, Dual Pentium 4 – 3 GHz, 512 MB, 1 Gbps
LB:	Shuttle xpc, Dual Pentium 4 – 3 GHz, 512 MB, 1 Gbps
Proxy 1:	Dell Inspiron 9300, CPU 1.6 GHz, 512MB, 1 Gbps
Proxy 2:	HP DX2000, CPU 2.66 GHz, 128MB, 1 Gbps
Proxy 3:	HP DX2000, CPU 2.66 GHz, 128MB, 1 Gbps
Proxy 4:	HP DX2000, CPU 2.66 GHz, 128MB, 1 Gbps
Server:	Shuttle xpc, Dual Pentium 4 – 3 GHz, 512 MB, 1 Gbps

Table 4.1: Specifications of the machines used in the experiments.

The client sends requests to the virtual service IP address (10.0.0.10), which is assigned to the LB and all the proxies. Since the load balancer is on the same LAN as the proxies in our setup, this can lead to a conflict when the client sends out an ARP for the VIP. There are various ways to deal with this issue [10], for example, by making sure that only the LB responds to the ARP requests for the VIP, and the proxies ignore those requests. Here, we have chosen a simple solution of creating a static ARP entry on the client, mapping the service VIP (10.0.0.10) to the MAC address of the LB.

In a real system, if separate machines are used as proxies, they will typically be multi-homed with separate interfaces for the server and the clients for security reasons. However, that is not a concern in our experiments.

4.8.2 Proxy scalability

In these experiments we measure the time taken to transfer files of different sizes from clients to a server using the HTTP PUT method. 100 simultaneous transfers are done in each case. A `C` program was written to simulate 100 clients. It calls `fork()` 100 times and each of the 100 children connect to the server and sent the PUT request followed by the data.

We chose two files sizes: 10 MB and 50 MB. The reason behind these choices is that these could represent the sizes of music, picture, or video files that a user may upload to a server through her web browser. Thus, in all, for 100 connections, 1 GB and 5 GB are transferred in the two cases, respectively.

For both of these file sizes, we conduct experiments for two different configurations: (1) a base configuration (C–S), where the client machine is directly connected to the server through the switch; and, (2) proxy configurations (C–LB–nP–S), where the client-server connection goes through a L3 load balancer (Section 4.5.2) and a proxy (Section 4.6). The base configuration is used as a baseline for comparing the performance of the proxy architectures. In the proxy configurations, the number of proxies is varied from 1 to 4.

Since we are interested in measuring the scalability of the proxies, we monitored the CPU and memory usage of the client, server, and the load balancer machines during each of the experiments to make sure that none of them become a bottleneck. Note that to prevent such a situation from occurring we have used machines with more powerful CPUs as the client, LB and server. This also makes the configuration more representative of a real-life situation, where there will be a multitude of clients and servers.

`gnome-system-monitor` was run on these machines to observe the graph (updated every sec) of their CPU and memory usage.

Furthermore, we took at least three measurements for each experiment, excluding the first run which was discarded to minimize the effect of cache misses. An Apache [7] web server was run on the server machine. A simple CGI script was written to handle the PUT requests. It looks at the "Content-length" tag in the HTTP request header, reads those many bytes, and discards them.

4.8.2.1 Discussion of results

The results are summarized in Tables 4.2 and 4.3 for 100 concurrent PUTs of sizes 10 MB and 50 MB, respectively. The average, min and max times in these tables indicate the data transfer times (excluding connection setup time) for a single connection and are averaged over three runs. Since the data transfer for the 100 connections is started simultaneously, the total time taken is the same as the time taken by the slowest connection (max time). The corresponding network throughputs achieved are also listed. The last column in the tables shows the impact on the network throughput of increasing the number of proxies. When the number of proxies is increased from 1 to 2, ideally, if the system scales linearly by the same factor, the throughput should also double. In our case, the throughput increases by about 74% (from 125 to 217 Mbps, and 126 to 219 Mbps, in the two cases, respectively). Similarly, with three–proxy configuration, the percentage increase over two–proxies configuration should be 50%, but it is actually around 30% and 32%. Finally, for the four–proxy configuration, the increase is 18% and 23%. Thus, the increase in network throughput with the increase in the number of proxies, shown in Figure 4.9, is almost linear up to at least four proxies. As the number of proxies is further increased, the throughput is likely to hit a knee and flatten out. In the future, we would like to add more proxies and determine how close the network throughput of the proxy architecture can get to that of the base configuration (client

and server connected directly).

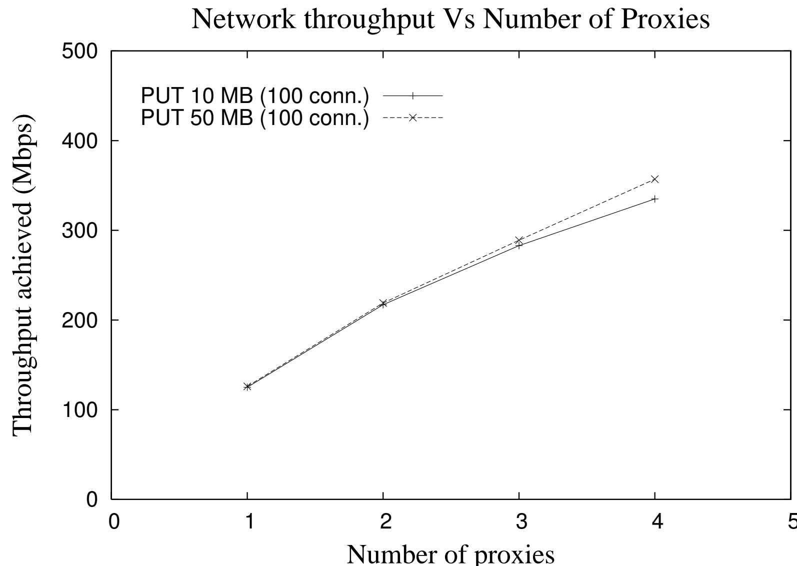


Figure 4.9: The plot shows the increase in the network throughput as more proxies are added.

We want to emphasize the scalability advantage of a distributed TCP splice over a non-distributed TCP splice. If the traffic on an existing persistent TCP connection, that is traditionally spliced (non-distributed) through a proxy, increases, there is no way to spread the increased load to other proxies, even if other proxies are present. With distributed splice, however, workload of existing TCP connections can be spread across other proxies.

Config.	Time Taken (sec.)			Throughput (Mbps)	Throughput increase over previous proxy config.
	Ave.	Min	Max		
C–S	8.14	5.38	8.93	895	n/a
C–LB–P–S	59.56	43.74	64.17	125	n/a
C–LB–2P–S	34.27	24.55	36.90	217	73.6%
C–LB–3P–S	24.74	18.22	28.25	283	30.4%
C–LB–4P–S	20.47	14.91	23.88	335	18.3%

Table 4.2: Experimental results for 100 concurrent client PUT requests of a file of size 10 MB

Config.	Time Taken (sec.)			Throu-ghput (Mbps)	Throughput increase over previous proxy config.
	Ave.	Min	Max		
C–S	43.10	38.31	44.87	892	n/a
C–LB–P–S	307.25	266.25	318.73	126	n/a
C–LB–2P–S	177.12	160.49	182.94	219	74.2%
C–LB–3P–S	129.19	107.86	138.25	289	32.3%
C–LB–4P–S	106.87	92.83	112.07	357	23.4%

Table 4.3: Experimental results for 100 concurrent client PUT requests of a file of size 50 MB

4.8.3 Proxy Failovers

The objective of these experiments is to show that the impact of a proxy failure on the users is negligible. In these experiments, the client sends 100 simultaneous PUT requests to the server, similar to the experiments in the previous section. Note that these experiments are done with a 100 Mbps switch, and, hence, the network throughputs are lower; however, this does not impact the results of these experiments.

Each request transfers 10 MB to the server. Experiments are performed with a two proxy configuration (C–LB–2P–S). A proxy is failed during the transfer by disabling its Ethernet interface. This failure is detected by the load balancer and it subsequently stops sending packets to that proxy. The failure detection time depends on the frequency of the heartbeat (HB). We conducted experiments with three different values of the HB frequency: 5s, 1s and 200ms. UDP is used for the HB, and the LB decides that a proxy has failed if it misses three HB in a row. In that case, it removes the failed proxy from its list of active proxies. Thus, for a HB interval of t sec., a failure is detected in between $2t$ and $3t$ sec.

Since after a failover, only one proxy is available for the remaining data transfer, the total time taken also depends on when the failure occurs after the start of the transfer. If the failure occurs right after the transfer starts, then the total time taken is expected to be closer to that of the one proxy configuration (C–LB–P–S). On the other

hand, if the failure occurs towards the end of the transfer, the total time taken is likely to be closer to that for the two proxy configuration (C–LB–2P–S). Thus, to remove this additional variable, and to be able to meaningfully compare experiments with different HB frequency, we fail a proxy after half of the average non-failure transfer time in these experiments. For 10 MB, the non-failure case takes about 90 sec., so we failed a proxy at about 45 sec. after the start.

4.8.3.1 Discussion of results

The results are summarized in Table 4.4. As expected, the average transfer times are longer for larger HB intervals. If the failures were to be detected instantaneously, the time taken should approximately be an average of the time taken with the C–LB–P–S and C–LB–2P–S configurations.

HB Interval	Time (in sec.)					Network Throughput (Mbps)
	Avg.	Min	Max	Std. dev	Total	
no failure	79.76	46.35	89.37	8.18	89.46	93.76
5 sec	90.54	54.66	142.07	8.86	142.14	59.02
1 sec	85.69	55.88	97.88	5.51	97.95	85.64
200 ms	82.20	45.30	94.42	5.70	94.48	88.79

Table 4.4: Experimental results for 100 concurrent PUTs of 10 MB with a proxy failure during the transfer.

For 1 and 5 sec HB intervals, it takes 2.5 and 12.5 sec on average to detect failure. During this time half the packets sent out by the client are dropped. This may cause the client to invoke TCP congestion control. We also calculated the network throughput achieved in each of these cases. The network throughput, of course, decreases when there is a proxy failure. This is because of packet loss during failure detection, fewer number of usable proxies for the remaining data transfer, and, if the failure has caused enough packet loss, TCP's recovery from congestion control. However, we do observe that the throughput remains relatively large when the HB interval is short. The best

results obtained are for HB interval of 200ms, where the failure is detected between 400 and 600ms. The average network throughput in this failure case is only 5.3% less than the no-failure case.

4.8.4 TCP Split Splice

The objective of these experiments is to show the advantage of TCP split splice, where the server performs the TCP splice header transformations on the response packets and sends them directly to the client. As before, 100 concurrent connections are created by the client, and, the HTTP GET method is used to download files from the server. Similar to the earlier experiments, files of two different sizes are transferred: 10 MB and 50 MB.

Three different configurations are used. (1) C–S, where the client directly connects to the server; (2) C–LB–P–S, where one proxy is used; other than the reversal in the direction of the data, this is identical to the PUT experiment performed with this configuration in Section 4.8.2; and, (3) C–LB–P–S*, which is similar to the previous configuration with the difference that split splice kernel module is installed on the server.

4.8.4.1 Discussion of results

Tables 4.5 and 4.6 show the performance measured for 100 concurrent GETs of sizes 10 MB and 50 MB, respectively. The results show that with split-splice (C–LB–P–S*), we get throughputs that are almost indistinguishable from those of the base case (C–S). This is to be expected since the data is directly sent to the client from the server. Furthermore, split splice is most efficient when bulk data has to be transferred from a server to a client.

Also, we observed that the CPU utilization of the server increases substantially when it is performing split splice. If a server does not have adequate CPU resources, the transfer rate can get CPU bound. However, we did not encounter this. While

Config.	Time Taken (sec.) Average	Min	Max	Throughput (Mbps)
C–S	8.14	5.38	8.93	895
C–LB–P–S	59.56	43.74	64.17	125
C–LB–P–S*	8.69	4.51	9.23	866

Table 4.5: Experimental results for 100 concurrent GET of 10 MB with **split splice** at the server.

Config.	Time Taken (sec.) Average	Min	Max	Throughput (Mbps)
C–S	43.10	38.31	44.87	892
C–LB–P–S	307.25	266.25	318.73	126
C–LB–P–S*	43.26	35.34	44.99	889

Table 4.6: Experimental results for 100 concurrent GET of 50 MB with **split splice** at the server.

performing split splice, our server was running at about 70% CPU utilization (one CPU was at about 100% and the other at about 40%).

4.9 Summary

User applications that usually run on a machine locally are beginning to be offered remotely through a web browser. Furthermore, users are increasingly using servers to store large amounts of data, such as images and videos. These newer applications require relatively long-duration, stateful client server sessions, and the amount of data flowing from the clients to the servers is relatively large. This work proposes three enhancements to the TCP splicing mechanism which then form the basis of a fault-tolerant, flexible web server architecture described in the next chapter. Performance measured from a prototype implementation in Linux is encouraging. It supports the key objectives that the proposed architecture is scalable and fault-tolerant.

Chapter 5

Fault-Tolerant Web Server Architecture

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 we present detailed performance results for a PHP-based webmail application deployed over this architecture.

5.1 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 an 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 [28] 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 chapter, 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 following 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 chapter is based on some of our work on enhancements to TCP splicing mechanism [49] (see Chapter 4 and systems architectures for transactional network interfaces [50] (see Chapter 6). There are three unique contributions of this work. 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 [63], an open-source webmail client) over this architecture.

The rest of this chapter is organized as follows. Section 5.2 describes some of our earlier work on which this work is built as well as some related work. Section 5.3 provides a high-level description of our complete system architecture. Section 5.4 describes important details including TCP re-splicing, transactionalization and tagging, and recovery mechanisms. Section 5.5 describes some salient features of our prototype implementation. Section 5.6 describes the details of RoundCube Webmail deployment over our architecture, and the performance measured from this deployment under many different operating scenarios. Finally, Section 5.7 summarizes the chapter.

5.2 Background

5.2.1 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 [44, 70] 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 [62, 61, 22]. 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.

5.2.2 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 [49]: (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.

5.2.3 Related work

Our architecture is most related to FT-TCP [4, 80], ST-TCP [47], Backdoors [16, 72] 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 [16, 72] 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. For further discussion of related work, the reader is referred to Chapter 2.

5.3 System architecture: An Overview

Figure 5.1 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.

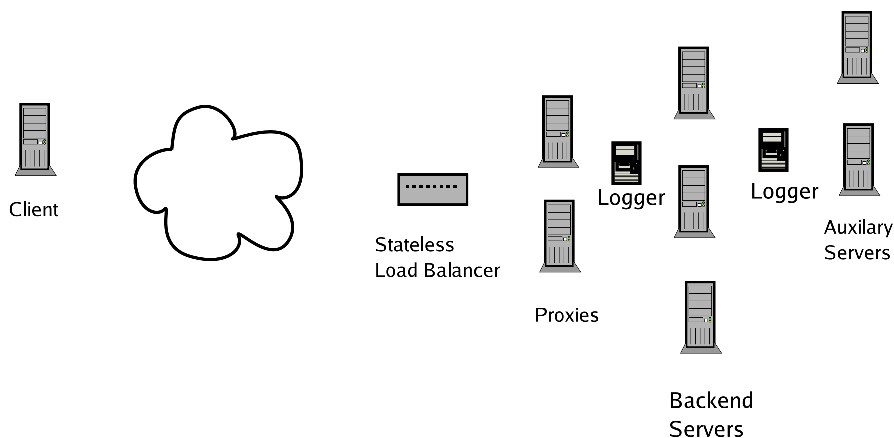


Figure 5.1: Components of our Web Server Architecture.

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 [49]. In fact, multiple proxy failures are handled similarly. In this work, 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 5.1, 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 [49]. 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 restarted 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.

5.3.1 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 work.

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.

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 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.

5.4 Detailed Design

5.4.1 Normal Operation

Steps required for handling a client request are as shown in Figure 5.2. 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 [49].

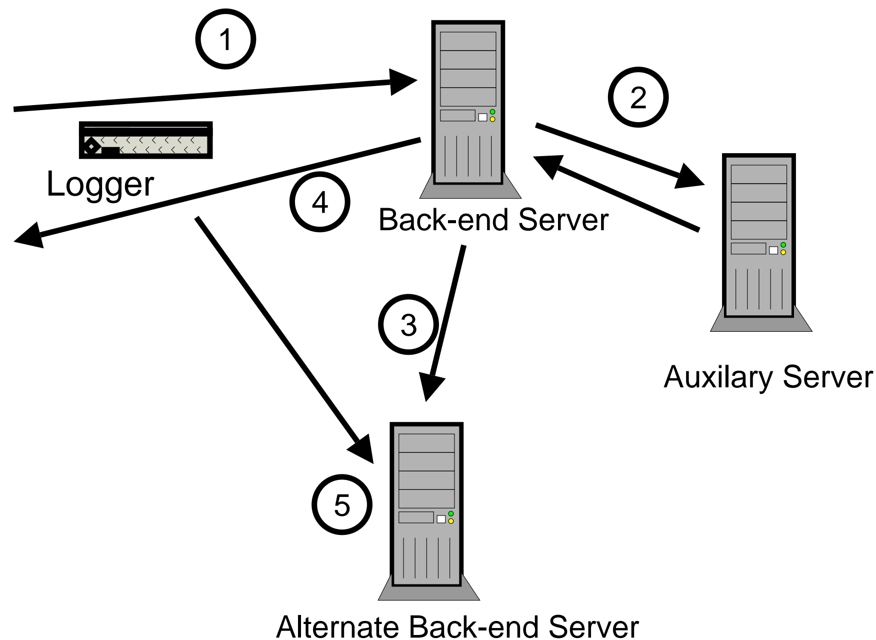


Figure 5.2: Sequence of steps required in handling a client request.

(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).

5.4.2 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_{T_L} Ack corresponding to the last response byte in transaction T_L.

5.4.3 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.

$$Ack_{CL} \leq Ack_{T_L} + p \tag{5.1}$$

Note that Equation 5.1 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,

$$T_{SP} = T_L + 1 \tag{5.2}$$

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.

$$T_R = \begin{cases} T_A + 1 & \text{if } T_L \geq T_A \\ T_L + 1 & \text{otherwise} \end{cases} \tag{5.3}$$

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.

5.5 Implementation

We implemented a prototype of our server fault-tolerance architecture in Linux. We had earlier made some enhancements to TCP splice [49] 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 [50] 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 [53] 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 5.3.1. 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.

5.6 Experiments and Performance Evaluation

To provide a proof of concept, we demonstrate our architecture on a real-life Web mail application called Roundcube webmail [63], 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 [23] to connect to the email store servers. Compared to traditional webmail clients, Roundcube and other similar AJAX-based [2] 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 net-

work 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 [14] 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.

5.6.1 Experimental Setup

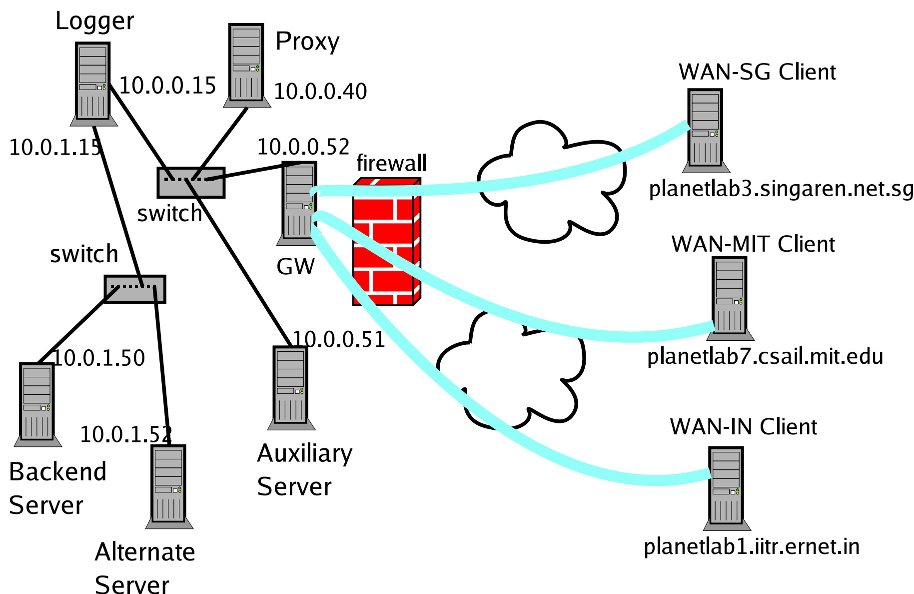


Figure 5.3: Experimental setup.

The experimental setup is shown in Figure 5.3. At the server end, the machines

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 5.1: 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.

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.

5.6.2 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 5.1. 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` [25] 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 perform 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 issues 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.

5.6.3 Results and Discussion

The results of our experiments for Action 1 and Action 2 are summarized in Tables 5.2 and 5.3, 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.

	Average Time Taken (sec)						
	Traditional Architecture			Our Architecture			Overhead during
	No-Failure	With Failure	Failover	No-Failure	With Failure	Failover	Normal Operation
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 5.2: Time taken for performing one run of Action 1: Connecting to the login screen.

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 re-splicing code are correctly

	Average Time Taken (sec)						
	Traditional Architecture			Our Architecture			Overhead during
	No-Failure	With Failure	Failover	No-Failure	With Failure	Failover	Normal Operation
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 5.3: Time taken for performing one run of Action 2: Logging in; drafting and sending an email; and, logging out.

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 $T_L = T_S = T_A$ (using terminology from Section 5.4.2). 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.

5.7 Summary

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 chapter, 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 appli-

cation. 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.

Chapter 6

Systems Architectures for Transactional Network Interface

Systems such as software transactional memory and some exception handling techniques use transactions. However, a typical limitation of such systems is that they do not allow system calls within transactions. This is particularly true for system calls that interact with file systems, devices, and the network. This chapter describes systems architectures that can be used to extend a transactional system to allow network read/write system calls within a transaction. This is done by delaying the sending of network bytes to a peer until a transaction is committed, and implementing a rollback mechanism in case a transaction aborts. Three different architectures, one transport-layer and two application layer, are proposed to incorporate this extension. The chapter discusses the advantages and limitations of each of these architectures. Prototypes of each of the three architectures have been implemented. This chapter describes the design and implementation of these prototypes, and provides an extensive performance evaluation under many different scenarios, including LAN environment, WAN environment (PlanetLab), communication-intensive transactions, and computation-intensive transactions.

6.1 Introduction

Transactional constructs are being implemented in modern programming languages to provide infrastructure support for systems such as software transactional

memory (STM) and some exception handling techniques. STM [67, 45] provides a simple and effective alternative to various lock-based synchronization mechanisms used in concurrent programming. It converts each critical section of the code, marked by the programmer, into a transaction. The transactions proceed without any locks. At the end of a transaction, if more than one concurrently running transactions have made conflicting memory accesses, only one transaction is allowed to proceed. The remaining are aborted and restarted.

One important restriction in such transactional systems is that they do not allow system calls within transactions. The key reason for this restriction is that it is very difficult to roll back certain system calls. This is particularly true for systems call that interact with file systems, devices, and the network. The fact that state information may escape to external entities such as the disk, devices, or the network makes rollback particularly challenging in these situations. However, an inability to include system calls within transactions severely limits their applicability. With communication networks being increasingly pervasive, more and more everyday computing applications involve some kind of network communication. As a result, current transactional systems that disallow network reads/writes with in a transaction do not provide adequate support for most current computing applications.

In this work, we address this limitation of current transactional systems by incorporating network read/write system calls within a transaction, and providing support for rollback in case the transaction aborts. The key functionality needed to provide this support is to temporarily suppress the sending of the network bytes originating from a transaction until that transaction commits. These bytes are discarded and an appropriate rollback mechanism is incorporated in case the transaction aborts. Providing such a support is difficult because of two reasons: (1) To be effective, no changes should be required at the peer (client); and (2) The performance of the server application should not suffer, particularly during normal operation when most of the transactions commit

the first time.

We propose three different system architectures. All three architectures provide support for including network reads/writes within a transaction, and rollback when a transaction aborts. One of these architectures, transport layer architecture, operates at the transport layer, and the other two, two-connection application layer and single machine applications layer architectures operate at the application layer. All three architectures satisfy the following important properties:

(1) They are completely transparent to the networking peer (client), i.e. absolutely no changes are required at the peer.

(2) No OS changes are needed at the server, and only minimal OS changes are needed at the logger.

(3) Only minimal changes are needed at the server application.

(4) There is minimal overhead under normal scenario, when the number of transaction aborts is very low. The transactional rate achieved in this scenario is comparable to that of non-transactional sends and receives.

We discuss the advantages and limitations of each of these architectures. To evaluate these architectures, we have implemented their prototypes. The chapter describes the design and implementation of these prototypes, and provides an extensive performance evaluation under many different computing scenarios, including LAN environment, WAN environment (PlanetLab), communication-intensive transactions, and computation-intensive transactions.

The performance results show that the overhead of using a transaction network interface is relatively insignificant. In fact, for communication-intensive transactions, the overhead becomes negligible. Also, the performance approaches that of the non-transactional transfer rate as the number of aborts goes to zero. In terms of perfor-

```
atomic {
  x = y;
  ...
  read(sock, buf1, SIZE);
  ...
  write(sock, buf, SIZE);
  ...
}
```

Figure 6.1: Use of socket read/write within an atomic STM block.

mance, no single architecture is clearly superior to the others. However, there are other design constraints that may make one architecture preferable to another for a particular situation.

This chapter is organized as follows. In the next section, we consider STM as an example system that would benefit from a transactional network interface. This is followed by a section on the system architecture of the proposed mechanisms for providing a transactional network interface. Implementation details of our prototypes of the proposed architectures comprises Section 6.4. The performance results of our experiments are presented in 6.5, which is followed by a summary.

6.2 Example Application of a Transactional Network Interface

Figure 6.1 illustrates an example of a subprogram where STM is used. The programmer has designated a set of statements containing network read and write as a transaction by simply enclosing them within an **atomic** construct.

In order to support network read/write operations in a transaction, the following functions are provided to the STM system.

- **tcp_tx_start()** This function is called to start a transaction on a TCP connection.
- **tcp_tx_send()** This function is used to send transactional data on a TCP

socket.

- **tcp_tx_receive()** This function is called to receive transactional data in a TCP socket.

- **tcp_tx_commit()** When a transaction is complete and ready to commit, this function is called to commit the sent and received data on a TCP connection.

- **tcp_tx_abort()** This function is called to abort an ongoing transaction on a TCP socket.

- **tcp_tx_validate()** While a transaction is ongoing, this function can be invoked to determine if some event has occurred that would require the transaction to be aborted.

The compiler re-writes an atomic block as a transaction as shown in Figure 6.2. Every memory read or write access is converted into a function call. These functions record read and write access information, which is later used to determine if a transaction can successfully commit. In case commit fails, it is rolled back and restarted.

An important feature in all three system architectures is that absolutely no changes are needed at a peer. This includes no new software installations and no software upgrades at the peer. As a result, transactions that require a **response** from a peer before the transaction commits are not supported. For example, a transaction that sends some data to a peer and needs a response before the end of the transaction will lead to a deadlock. This issue can be addressed by breaking up such a transaction into multiple smaller ones. Furthermore, in most scenarios a peer is a client and initiates a request and expects a response from the server—a case that does not lead to this problem.

Also, the process of establishing a TCP connection cannot be a part of a transaction in these architectures. However, if a FIN or RST is received from a peer during

```
do {
   stmStart(); // includes tcp_tx_start()

     temp = stmRead(&y);
     stmWrite(&x, temp);
     ...
     tcp_tx_receive(sock, tmpBuf1, SIZE);
     stmWrite(buf1, tmpBuf1, SIZE);
     ...
     stmRead(tmpBuf, buf, SIZE);
     tcp_tx_send(sock, tmpBuf, SIZE);
     ...

   if (stmCommit() == 0);
  // includes tcp_tx_commit()
     break;

} while (1);
```

Figure 6.2: Compiler generated code for the code segment shown in Figure 6.1

an ongoing transaction, it will not be treated any different from a data segment, and the application would need to handle such connection failures during a transaction.

All three architectures provide support for only flat transactions, i.e. no nested transactions are allowed. If there are any nested transactions, they would need to be converted into flat ones in order to use the extended STM support. It is important to note that nested transactions do not make much sense for TCP connections, where read/write operations are strictly sequential.

Finally, a TCP connection cannot simultaneously be part of more than one transactions. So, all accesses to a particular TCP socket must be serialized. In practice, it is unlikely that this will not already be the case, since multiple threads simultaneously writing to the same socket are likely to produce gibberish at the other end.

6.3 System Architecture

In order to provide a transactional network interface for TCP connections, we propose three different system architectures. The first one is a transport layer architecture (TLA, for short), and the other two are application layer based. We refer to one of the application layer architectures as two-connection architecture (2CA), since it requires two TCP connections, and the other as single machine architecture (SMA). The TLA and 2CA typically require a logger to temporarily store the network bytes being exchanged between the server and its peer. SMA does not require a separate logger.

The basic idea in all three architectures is that bytes sent out by the server application are not delivered to the peer until the corresponding transaction commits. Furthermore, the bytes destined to the server application are logged, so that those bytes can be resent without peer involvement when a failed transaction restarts. In TLA and 2CA, the transaction bytes (in both directions of the TCP connection) are saved at a separate logger. In SMA, they are saved in the main memory of the application server.

Notice that while TLA and 2CA require a separate logger, this logger can be shared with other systems that provide support for failure recovery, security related logging, etc.

6.3.1 Transport layer architecture

The transport layer architecture (TLA), shown in Figure 6.3, requires a logger that provides transport layer support for the management of transactions. The STM library shown in the figure contains the tcp_tx_* functions listed in Section 6.2. All TCP segments, in both directions, are routed through the logger machine, where they are intercepted and logged. A very important property of TLA is that the end-to-end TCP connection semantics are preserved between the peer and the server application. This is analogous to TCP splicing [44, 70, 49, 62] used in building high performance web servers.

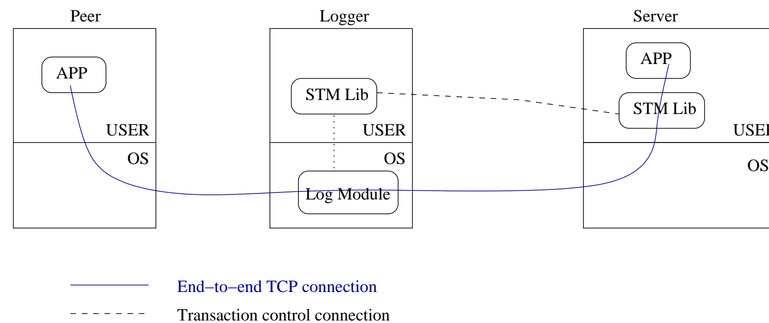


Figure 6.3: System architecture.

6.3.1.1 Control Connection

Both TLA and 2CA (described in Section 6.3.2) require a separate control connection between the server and the logger. This TCP connection is used for communicating transaction events, such as transaction **start, commit, abort,** or **restart**. The following messages are exchanged on this control channel:

- Requests (server to logger)
    - ∗ TX_START (startingByte, endingByte)
    - ∗ TX_COMMIT (startingByte, endingByte)
    - ∗ TX_ABORT (startingByte, endingByte)
    - ∗ TX_RESTART
- Responses (logger to server)
    - ∗ TX_SUCCESS
    - ∗ TX_FAILURE

The `startingByte` and `endingByte` arguments in the first three request messages are the offsets in the TCP connection stream where the corresponding action starts and ends, respectively. The request/response messages are synchronous, i.e. after sending a request, the server waits for a response before continuing. The two responses indicate whether the request sent succeeded or not.

6.3.1.2 Transaction processing

There are three different phases in processing of a transaction: (1) when a transaction is ongoing; (2) when a transaction is committed; and (3) when a transaction is aborted.

- **Ongoing transaction:** The server has sent a TX_START to the logger. The TCP segments that are received from the peer are saved at the logger, and then immediately sent on to the server. The segments that are received from the server are saved at the logger, but **not** sent to the peer. Note that these TCP segments are saved in the kernel, and so no kernel-to-user or user-to-kernel data copying is involved. Since the server to peer bytes are not immediately

forwarded to the peer until a transaction commits, the number of bytes that a server can send as a part of a single transaction is atmost the size of its TCP send buffer. Similarly, since the acks do not reach the peer during a transaction, the peer is also limited to sending at most a TCP send buffer worth of data as part of a single transaction. This is not a problem in practice, because TCP buffers are quite large, accommodating several megabytes of data per connection, particularly in newer systems equipped with gigabit network cards. Furthermore, the regular non-transactional network interface may be used for transferring bulk data, e.g very large files.

- **Transaction is committed:** The server sends a TX_COMMIT to the logger once it has successfully completed a transaction including all network operations, and is ready to commit. Since the control connection is separate from the end-to-end peer-server TCP connection, it is possible that the server TCP send buffers may still be holding (unsent) data belonging to the transaction that was just committed by the application. On receiving a TX_COMMIT, the logger sends out all the saved TCP segments destined for the peer. Recall that these segments were stored in the kernel, and so no user-to-kernel data copying is involved in this operation. The logger also purges these TCP segments after forwarding them to the peer, as well as purges the saved peer-to-server TCP segments related to the just committed transaction.

- **Transaction is aborted:** When the server application needs to abort a transaction, it sends a TX_ABORT to the logger, followed by a TX_RESTART as it prepares to re-execute the aborted transaction. When a transaction aborts, the logger needs to perform a number of tasks related to both the segments received from the server and those from the peer:

* **Server-to-Peer Segments:** Logger discards all the bytes that the server had sent as a part of the aborted transaction. Thus, these are the bytes that the server has sent to the peer, and the peer never receives them, and hence never acknowledges them. So, these bytes must be acknowledged by the logger. The logger acks these bytes by generating one or more *fake* acks and then purges the corresponding saved bytes. Note that acking and discarding these bytes creates a "hole" in the server sequence number space from the viewpoint of the peer. Thus, in order to keep this transparent from the peer, the sequence numbers on any subsequent segments sent to the peer on this connection are suitably adjusted. A corresponding adjustment is made to the acks going in the opposite direction. Again, this sequence number mapping in the TCP segments going in both directions is done in the kernel, analogous to TCP splicing [70].

* **Peer-to-Server Segments:** These are the bytes that the peer has sent to the server that were a part of the aborted transaction. Recall that these bytes were saved in the logger before being forwarded to the server. These bytes have to be re-sent to the server once the (aborted) transaction is re-started. The logger sends these bytes with **new** sequence numbers. Note that this entire process of restarting a transaction is completely transparent to the peer. As with server-to-peer TCP segments, the sequence number of subsequent peer-to-server TCP segments are suitably adjusted due to the insertion of these (re-sent) peer-to-server bytes. Also, a corresponding adjustment is made in the acks going from the server to the peer. Once again, this sequence number mapping in the TCP segments going in both directions is done in the kernel, analogous to TCP splicing [70].

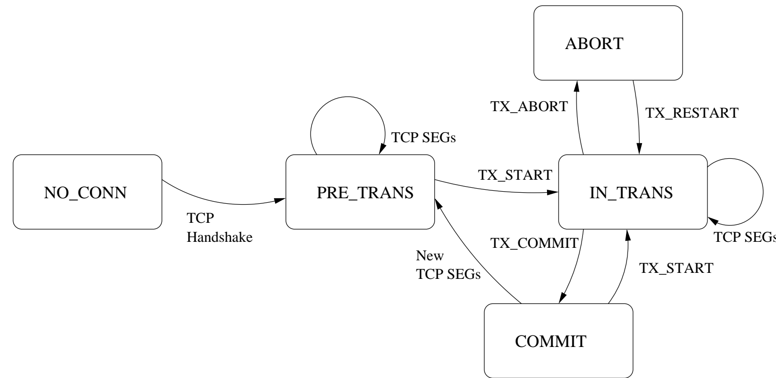


Figure 6.4: States assigned to a transactional TCP connection at the logger.

6.3.1.3 Logger State

Figure 6.4 shows the states a TCP connection is assigned at a logger in order to support transactional data on that connection. A server application can run any number of transactions on a TCP connection, as long as all these transactions are serialized, i.e. only one transaction can run on a connection at any instant.

After a TCP connection is established, the state of that connection is marked as PRE_TRANS and data passing on that connection is logged. This is done even though a TX_START has not yet been received from the server. The reason for this is as follows. When a server starts a transaction, it may read bytes, part of that transaction, that may already be in the TCP receive buffers of the server. Thus, it is possible that the initial transaction bytes from the peer may pass the logger before the server can indicate a TX_START. To address this scenario, the logger saves all segments – even those that are not a part of an ongoing transaction, since they may become part of a future transaction. Saved bytes that do not get associated with a later transaction are aged out and discarded. This aging out is done both on the basis of the amount of such data and the time duration for which it has existed on the logger. While in the PRE_TRANS state, the logger continues to save TCP segments that arrive. It transits to the IN_TRANS state when a TX_START message is received from the server. This

message also includes the starting TCP sequence number offsets, in both directions, for that transaction.

From the IN_TRANS state, the server can signal to commit or abort a transaction. The connection for a committed transaction stays in the COMMITTED state until all the segments related to the transaction are reliably received. The commit message from the server also includes TCP sequence number offsets marking the end of the transactions. If a segment with a sequence number beyond the end of the committed transaction is received, the logger transitions to the PRE_TRANS state.

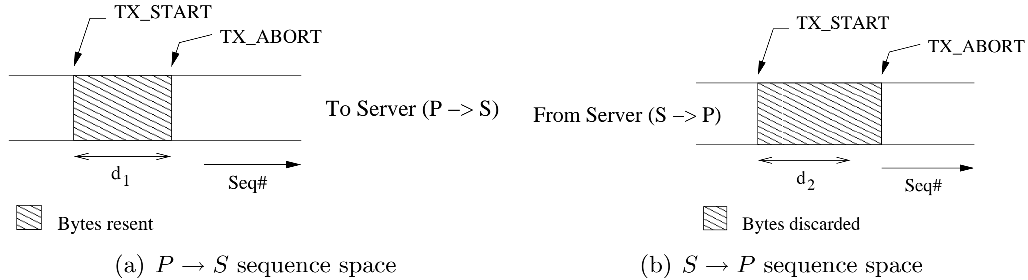
If a transaction aborts, the logger acks and discards server-to-peer bytes. As mentioned earlier, from the peer's perspective, this creates a hole in the server's sequence number space that needs to be plugged by suitably modifying the sequence and ack numbers of future segments. A similar adjustment is also required due to resending of bytes to the server. Figures 6.5(a) and 6.5(b) show the sequence number space of a server-to-peer TCP connection. Assume that the number of peer-to-server bytes resent is d_1, and the number of server-to-peer bytes discarded is d_2. Then for the peer-to-server ($P \rightarrow S$) direction, the new sequence ($S'_{p\rightarrow s}$) and ack ($A'_{p\rightarrow s}$) numbers can be computed as:

$$S'_{p\rightarrow s} = S_{p\rightarrow s} + d_1; \quad A'_{p\rightarrow s} = A_{p\rightarrow s} + d_2$$

Similarly, for the server-to-peer ($S \rightarrow P$) direction, the new sequence ($S'_{s\rightarrow p}$) and ack ($A'_{s\rightarrow p}$) numbers can be computed as:

$$S'_{s\rightarrow p} = S_{s\rightarrow p} - d_2; \quad A'_{s\rightarrow p} = A_{s\rightarrow p} - d_1$$

Once again, note that this is similar in concept to how sequence and ack number are modified in TCP splice [70].



(a) $P \rightarrow S$ sequence space

(b) $S \rightarrow P$ sequence space

Figure 6.5: Sequence and ack number adjustments are required due to transaction aborts.

6.3.1.4 TLA: Discussion

The main advantage of TLA is that it maintains the end-to-end TCP connection semantics between the peer and the server. It employs techniques similar to TCP splicing by performing all buffering and sequence number mapping in the kernel. As a result, based on the experience from TCP splicing [49], the performance overhead due the introduction of the logger is expected to be insignificant.

There are some concerns in using TLA. One is its impact on the calculation of RTT. Since the logger holds server segments until a transaction commits, the round trip time (RTT) computed by the server will be skewed, which could adversely impact the throughput between the server and its peer. However, the actual data transfer during a transaction is not very high (a few megabytes at the most). So, the impact of a somewhat skewed RTT is expected to be minimal. Also, as mentioned earlier, non-transactional network interface is used for large data transfers.

Another concern is related to the duration of a transaction. Since the data sent out during a transaction is not acknowledged until the transaction is committed, the TCP connection may fail if the transaction lasts too long. The server and peer TCP may generate re-transmissions, which although harmless from the point of view of correct functioning, can lead to some overhead. However, transactions are not expected to last very long in a practical setting, perhaps a few times the RTT at the most.

Finally, the amount of data sent/received during a transaction is bound by the size of the TCP send buffer on the server and the peer. As mentioned earlier, this is not a problem in practice, because TCP buffers are typically quite large, accommodating several megabytes of data per connection.

6.3.2 Two Connection Application layer architecture

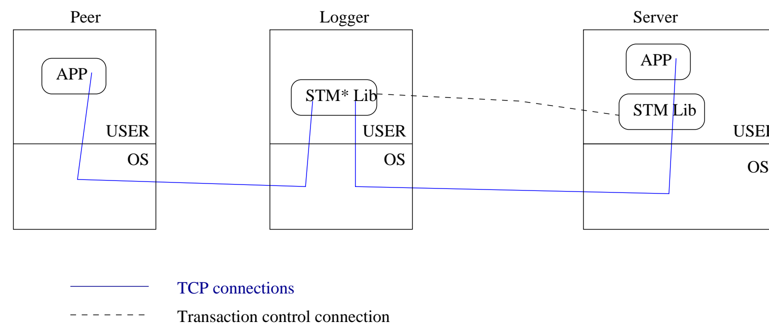


Figure 6.6: Application level architecture with two TCP connections.

The two-connection application layer architecture (2CA) is shown in Figure 6.6. Here the TCP connection between the peer and the server is broken up into two distinct TCP connections: (1) a TCP connection between server and the logger (logger—server TCP connection); and (2) a TCP connection between the logger and the peer (logger—peer TCP connection). Logically, the logger in this architecture operates in a very similar fashion as that in TLA. In fact, the control connection between the server and the logger and the messages exchanged on this connection (see Section 6.3.1.1) are identical.

The logger state machine is also almost identical to that of TLA shown in Figure 6.4. The key difference is that when a transaction is ongoing, the logger application saves the server and peer bytes in main memory. When the logger application receives bytes from the server, it saves them in main memory. Note that these bytes are acked immediately by the logger—server TCP on the logger side. When the logger application

receives bytes from the peer, it saves them in main memory, and sends them to the server via the logger—server TCP connection. When a transaction commits, the logger application sends out the server bytes to the peer via the logger—peer TCP connection. Similarly, when a transaction aborts, the logger application discards server bytes, and resends peer bytes to the server via the logger—server TCP connection. Conceptually, this architecture is very similar to that of a proxy application, reading data from one socket and writing it to the other.

6.3.2.1 2CA: Discussion

The main advantage of 2CA is that no OS changes are required at the logger. Furthermore, the amount of data sent during a transaction is not limited by the size of the TCP send buffers. There are two major drawbacks of 2CA as compared to TLA: (1) The end-to-end TCP semantics of the connection between the server and the peer are no longer preserved. The TCP bytes now logically go from the peer (server) to logger, and then from the logger to the server (peer). This is in contrast to TLA, where the TCP bytes are logically exchanged between the server and the peer. (2) 2CA is expected to be slightly inefficient, because it requires copying of data between kernel space and user space twice, once when the logger application reads data from a socket, and then again when it writes that data.

6.3.3 Single machine application layer architecture

The single machine architecture (SMA) is shown in Figure 6.7. In this architecture, a separate logger is not used. Instead, the function of the logger is performed by local library functions. The tcp_tx_send function, described in Section 6.2, saves bytes in main memory when the application calls this function to perform a write on a socket. Similarly, the tcp_tx_receive function saves received bytes in addition to passing them on to the server application. Only when a transaction commits do these functions send

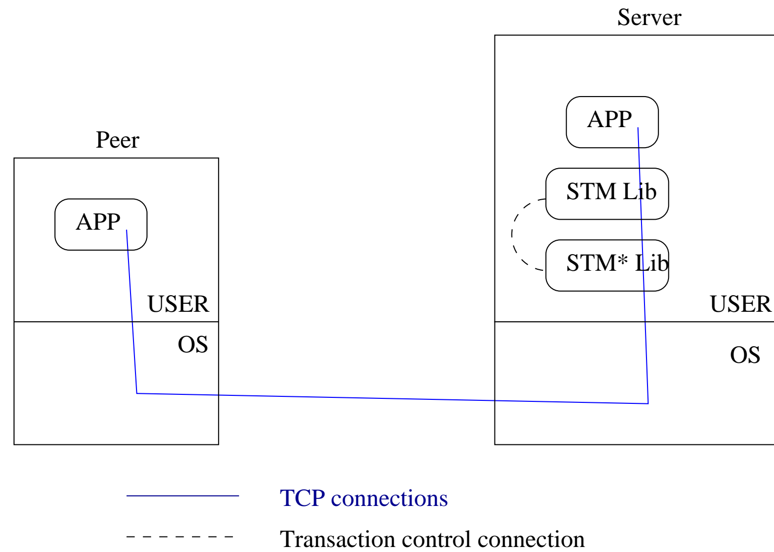


Figure 6.7: Application level architecture with no separate logger.

saved bytes to the peer, and purge the saved bytes, respectively. On a transaction abort, the tcp_tx_send function discards bytes without writing them to the actual socket, and the tcp_tx_receive function passes earlier transaction bytes again to the application. Transaction events such as start, commit and abort are communicated using an appropriate IPC mechanism. This architecture is similar in spirit to the one described in [31], which aims to support external I/O in atomic blocks.

The main advantage of this architecture is that it does not require any changes to the kernel, and since it does not split the server–peer TCP connection into two connections as is done in 2CA, it retains the the end-to-end semantics of a TCP connection.

The main drawback is that this architecture conceptually co-locates the functionality of the logger with the application server. This may be undesirable in many instances where server resources, such as memory and CPU are scarce. Furthermore, a failure of the logger code can potentially cause the application server to fail. Also, many users may already have a logger for other purposes that is shared by the application servers and may want to use that instead of installing additional software on every

application server.

6.4 Implementation Details

All three transactional network interface system architectures proposed do not require any changes in the network peer. Furthermore, no OS modifications are required on the servers. The TLA does require some OS changes, but those are limited to the logger, where a kernel module is used to intercept and manage the data belonging to the transactions. The 2CA and SMA do not require any OS changes on any machine. An important point to note is that the application on the server and the peer do not require any changes in any of the three architectures.

6.4.1 Transport layer architecture

The key element of TLA is the logger. We have implemented it as a Linux kernel module called `logmod` and installed it on the logger machine. As described earlier, all TCP segments destined to or sent from the server are routed through the logger machine. The `logmod` kernel module uses netfilter [53] to tap and process these segments.

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. Five such hooks are provided (See Figure 6.8). These hooks intercept packets and invoke any CB function that may be registered with that hook. When multiple CB functions are registered at a particular hook, they are invoked in the order of the priority specified at the time of their registration. 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.

The `logmod` kernel module is registered at the NF_IP_FORWARD hook which allows it to intercept all forwarded packets, in both directions. It discards packets not destined for or not originating from the service IP address and port number of the

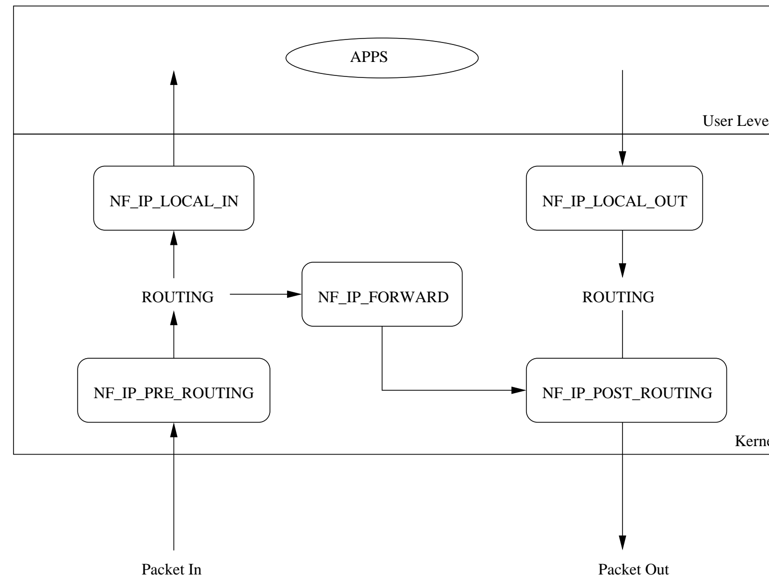


Figure 6.8: Hooks in netfilter.

server.

A user level process on the logger machine listens for a TCP connection from the server. This connection serves as the control connection on which the server signals transaction **start, commit, abort,** or **restart**. These are synchronous messages, i.e. after sending them, the server waits for a `SUCCESS` or a `FAILURE` response before proceeding further. The user level process uses `setsockopt` to communicate these events and the associated data to the kernel module.

`logmod` implements the state machine shown in Figure 6.4. It saves incoming TCP segments in a linked list of `sk_buff`, using a pair of lists – one for each direction – per connection. Only segments that contain data are stored in the list. In the case of ack-only segments, only the last such segment is saved. This does not lead to any loss of information, since TCP acks are cumulative.

For each direction in a connection, a set of state information, mainly related to sequence numbers, is kept in a data structure called `struct conn_info` as shown in Figure 6.9. This includes a pointer to the last ack-only segment received. This segment

is copied (`skb_copy()`) and used to generate a fake ack. The sequence and ack numbers are changed to the latest, the checksum is recalculated and the ack is sent off to the server.

As mentioned earlier, once an abort has occurred the sequence and ack numbers of any segment relayed by the logger between the server and its peer need to be adjusted. The amount that they need to be adjusted by is saved in `seqDiff`. After the sequence and ack number modification, the new TCP checksum is computed using incremental checksum update [60]. This allows the new checksum to be computed based only on the old and new value of the field that is modified and the old checksum. In particular, it does not require the recomputation of the entire checksum, and as a result, is significantly faster. This results in substantial savings, especially in case of TCP since its checksum includes the data in addition to the header.

A number of improvements can be made to the kernel module code to make it more efficient and generic. Right now, it handles only one TCP connection. This can be easily fixed by creating a hash table to keep track of all the existing connections. The peer IP address and port number would be convenient hash keys for quick mapping of incoming segments to their connections. Another potential improvement is a use of finer grained locking.

In the code, it is assumed that the byte in the TCP stream where an abort happens is at a segment boundary, i.e. it is the last byte in a segment. This assumption is only made for ease of coding the prototype, since it precludes cases where partial segments may need to be discarded by the logger on an abort. Although the latter case is not handled in the code, it is detected and flagged. In our experiments described later, the server performs sends in the multiples of 1460 bytes, which is a typical TCP MSS. Because of this, we did not encounter a case where an abort occurs in the middle of a segment. We might have encountered it if the path MSS of the connection was different than assumed. Note that it is relatively simple to not require this assumption,

```
struct conn_info {
    struct sk_buff_head  segQ;
          // segment linked-list
    __u32                startSeq,
         //tx start seq#
                         endSeq,
// tx end seq# (if committed)
                     abortSeq;
// tx abort seq# (if aborted)
    struct sk_buff      *lastAckSeg;
           // last ack only segment
    __u32                lastSeq,
   // last seq# seen
                         lastAck;
// last ack seq# seen
    __u32                isn;
// conn initial seq#

    // info for seq num mapping,
// required after an abort
    __u32                seqDiff;
}
```

Figure 6.9: Data structure for storing state information (mainly sequence number related) for each direction of a TCP connection.

and more importantly, it has no impact on the performance results.

The `logmod` Linux kernel module is about 1300 lines of `C` code that was tested with Linux 2.6.12 and is available at `http://magellan.colorado.edu/~marwah/logmod.tgz`

6.4.2 Application layer architectures

The implementation of the 2CA and the SMA is relatively straightforward. In 2CA, a logger application accepts the peer TCP connection and establishes a second connection to the server. It acts similar to an application-level proxy. Additionally, a control connection is established between the logger and the server. This is identical to the one used for the transport layer architecture. Also note that the server and peer applications do not require any changes. The logger application uses non-blocking I/O to monitor the three sockets, namely, the server socket, the peer socket and the control connection socket. It uses a large `char` array to save bytes read from the server, and on transaction commit, writes them out to the peer socket. For the remaining bytes in the transaction (note that signaling of commit is done on a separate channel and does not imply that TCP has actually transferred all the bytes), the logger application directly writes them to the peer socket without having to save it them intermediately.

6.5 Experiments and performance evaluation

We conducted a series of experiments over networks with diverse characteristics to evaluate the performance of the three proposed transactional network interface architectures, namely, TLA, 2CA and SMA. We measured five important performance characteristics in our experiments: (1) Measure the overhead of a transactional network interface as compared to a non-transactional one; (2) Compare the relative performance of the three architectures; (3) Measure the performance of the three architectures over WAN links with varied round trip times; (4) Measure the impact of percentage of transaction aborts in computational-intensive as well as communication-intensive transactions; and

(5) Measure the effect of transaction aborts on the performance of the system as the percent of transactions that abort is varied.

We performed the experiments in both LAN and WAN settings. The server and logger machines 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, the peer application was installed on a machine connected to the same LAN as the logger. In order to run our experiments over diverse WAN links, we used PlanetLab [14] nodes as our testbed. We placed the peer on two distinct sites: (1) a PlanetLab node at MIT (planetlab7.csail.mit.edu); and (2) a PlanetLab node in Singapore (planetlab3.singaren.net.sg). These locations were chosen, since they provide a wide variety of round trip times (RTT) between the server in Colorado and the peer. The RTT is about 69 ms for the machine at MIT and about 254 ms for the one in Singapore. Compared to these the RTT for the LAN scenario is about 0.25 ms.

6.5.1 Experimental Setup

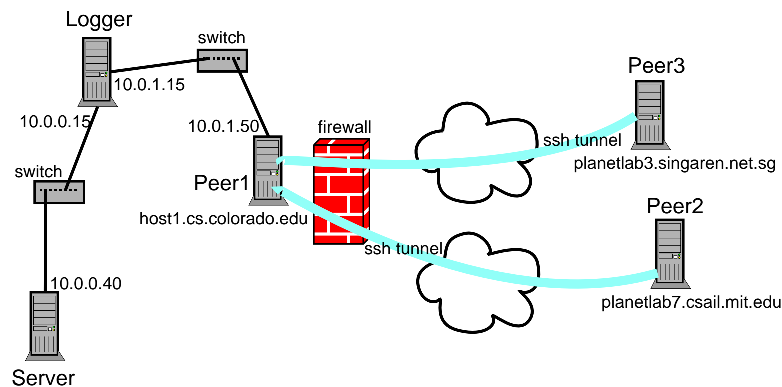


Figure 6.10: Experimental setup.

Like most organizations, the University of Colorado (CU) has a firewall which restricts outside machines from connecting to TCP ports on machines inside the firewall. Thus, a network peer on a PlanetLab node cannot initiate a TCP connection to a

server located inside the campus network. The only network port that is open to TCP connections originating from outside the campus network is the `ssh` port. We use ssh tunneling, supported in OpenSSH (a free version of ssh available on Linux platforms), to circumvent this problem. The peer-to-server TCP connection is tunneled through an `ssh` session between a machine inside the campus firewall and a PlanetLab node. The experimental setup is shown in Figure 6.10. The three locations where the server's peer is located during the course of the experiments are marked in the figure as Peer1, Peer2 and Peer3.

6.5.2 Experiments

A single run in our experiments consists of sequentially executing 100 transactions. A transaction is comprised of a request (a few tens of bytes long) sent to the server on a TCP connection, followed by a response of size 100 KB. The peer performs an active connect to establish a TCP connection and all transactions are sent on the same connection. From the server's perspective, a transaction starts when a request arrives and commits when it has generated and sent the entire response. We simulate an STM system on the server by performing aborts on a certain percentage of the transactions. We abort a transaction at most once, i.e. all transactions that abort on the first attempt succeed on the second. Since the point where a transaction aborts influences the performance of the system, all transactions that abort do so exactly mid-way in the transaction, i.e. when the server application has sent out half the output response. Furthermore, the actual transactions that abort in a run are chosen at random.

We assume that the computational cost of a completed transaction is 30 ms and that of an abort is half of that, i.e. 15 ms. This cost is implemented by using the `nanosleep()` system call (which on most Linux systems has a resolution of about 10 ms) on the server. Note that this system call only guarantees that the sleep time will be **at least** the amount passed as an argument. In practice, we found that for the values

we used, the actual sleep time is routinely a few ms over.

We run experiments for a range of first-attempt commit percentages, i.e. commits that occur without any aborts. In short, we also refer to this simply as commit percentage. We start with 0%, which corresponds to all transactions aborting in the first attempt before committing. The experiments are repeated at every interval of about 20%, with the final run at 100% commit, which corresponds to all transactions committing in the first try. Furthermore, we took at least three measurements for each run, excluding the first run which was discarded to minimize the effect of cache misses.

We run experiments for three different scenarios: (1) LAN, with the peer on the local LAN; (2) WAN1-MIT, with the peer located at MIT; and (3) WAN2-SG with the peer located in Singapore. For each of these scenarios, we conducted experiments for all three architectures, and for each architecture, six different values of commit percentage (ranging from 0% to 100%) are considered.

Regular, non-transactional network interface is also used in each scenario to transfer exactly the same request/response data as is done in the transactional architectures. The measurements obtained are used to determine the overhead of the three transactional network interface architectures.

6.5.3 Discussion of results: LAN Scenario

	TLA				2CA				SMA			
		Time per tx (ms)				Time per tx (ms)				Time per tx (ms)		
% Commit	tx/sec	ave	min	max	tx/sec	ave	min	max	tx/sec	ave	min	max
0	16.5	60.6	59.7	61.5	16.4	61.0	60.0	69.9	16.8	59.6	58.4	60.6
16	17.4	57.6	41.9	62.4	17.3	58.0	41.6	62.3	17.6	56.9	42.0	60.8
41	18.8	53.3	41.9	61.5	18.7	53.5	41.8	63.1	19.1	52.5	41.9	60.5
57	19.8	50.5	41.9	78.1	19.9	50.5	41.8	62.3	20.1	49.8	41.9	60.6
84	22.1	45.4	41.7	61.3	21.9	45.7	41.8	61.9	22.2	45.1	41.8	60.5
100	23.7	42.2	41.7	44.3	23.4	42.8	41.8	44.7	23.7	42.3	41.8	43.1

Table 6.1: Average transaction rate for the three architectures in the LAN scenario. Also listed are the average, minimum, and maximum times taken for a transaction.

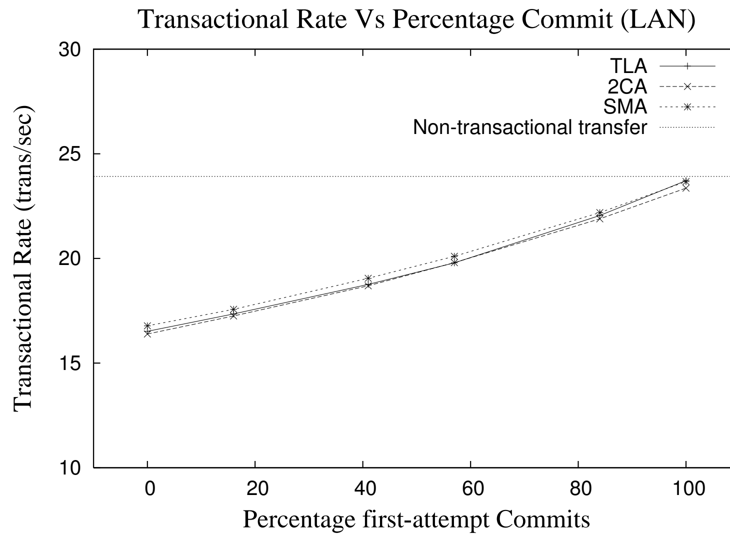


Figure 6.11: Transactional rate versus percentage of first-attempt commit transactions for the **LAN scenario**.

The first set of experiments consisted of running the transactions for the LAN scenario, with the peer on the same LAN as the logger. The logger is multihomed and as can be seen in Figure 6.10, all packets between the peer and the server must pass through the logger.

The transactional rates for the three architectures under different commit percentages are plotted in Figure 6.11. The graph shows that the transactional rate increases with increase in the percentage commits. As the number of aborts decline, the resulting saving in computational and communication costs are reflected in the increasing transaction rate. From Table 6.1, which summarizes the results, we see that TLA at 0% commit has an average transaction time of 60.6 ms. Out of this the computational cost is at least 45 ms (15 ms for the failed attempt plus 30 ms for the committed transaction) and the rest, 15.6 ms, is the communication cost. We calculate the computational and communication costs for all abort percentages for TLA and plot it in Figure 6.12. This graph shows that in addition to the computation costs (which are linear by design), the communications costs also grow only linearly with increase in the number of aborted

transactions. Although, the data plotted is for TLA, this result is true for the other two architectures as well.

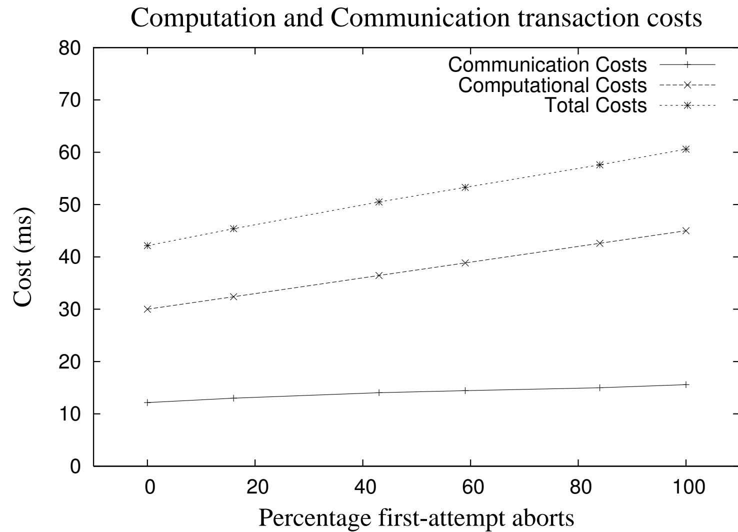


Figure 6.12: Communication and computational costs of a transaction for TLA in the LAN scenario.

When implemented without transaction semantics, the transactional rate is 23.92. This is the best rate that the transactional architectures can hope to achieve. From Figure 6.11, we see that as the percentage of first-attempt commits reaches 100%, the transaction rate of the three architectures approaches that of the non-transactional case. In fact, in the region close to 100%, the difference is quite low. This is especially important, because good transactional systems are engineered such that the abort percentage is low and that during normal operation the first-attempt commits are close to 100%. In such a scenario, the overhead of using one of the proposed transactional architecture is quite low.

The differences between the three architectures is not significant. Although SMA seems to perform marginally better, in percentage terms, the difference is negligible. Also, towards the 100% commit level, TLA and SMA seem to perform identical, while 2CA is marginally behind. It is possible that 2CA is slightly behind due to the fact that

it involves kernel-to-user and user-to-kernel space copying at the logger, however, the difference is small enough that this cannot be conclusively stated and requires further investigation.

The average transaction rates, together with the average, minimum and maximum time taken per transaction are summarized in Table 6.1.

6.5.4 Discussion of Results: WAN Scenarios

In order to measure the performance of our architectures with the peer separated from the server over a WAN link, we used a PlanetLab node at MIT and another in Singapore. Since the RTT's of the links to these machines are large, we expect the communication part of the total transaction time to be dominant. Taking one set of measurements for a WAN scenario for the three architectures with varying commit percentages takes at least a few hours (Note that we repeat each run three or more times and take the average.). For the WAN links, it took us several attempts of running the experiments in order to collect meaningful results due to significant temporal variation in the network characteristics of the links.

	TLA				2CA				SMA			
		Time per tx (ms)				Time per tx (ms)				Time per tx (ms)		
% Commit	tx/sec	ave	min	max	tx/sec	ave	min	max	tx/sec	ave	min	max
0	2.53	395	389	622	2.53	395	358	633	2.54	394	389	639
16	2.56	391	373	611	2.57	389	341	577	2.56	390	373	593
41	2.55	392	348	648	2.58	387	373	620	2.57	389	352	603
57	2.56	390	374	600	2.59	390	342	623	2.57	388	348	640
84	2.60	385	375	621	2.60	385	343	609	2.60	384	359	622
100	2.62	382	374	584	2.63	382	374	634	2.62	381	374	602

Table 6.2: Average transaction rate for the three architectures in the WAN1–MIT scenario. Also listed are the average, minimum, and maximum times taken for a transaction.

Figure 6.13 shows the average transaction rates for the MIT node (WAN1-MIT scenario). The increase in the transaction rate with the commit percentage is **not** very visible in this case unlike in the LAN scenario. Also notice that this is despite the fact

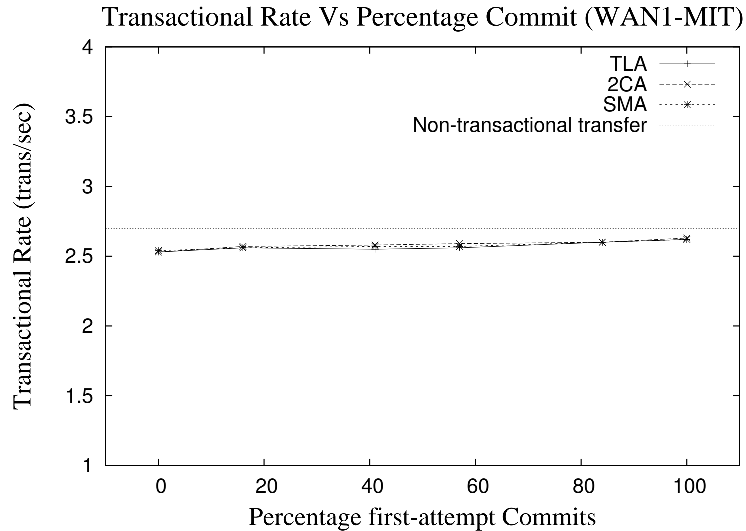


Figure 6.13: Transactional rate versus percentage of first-attempt commit transactions for the **WAN1-MIT scenario**.

that the y-axis in the WAN1-MIT case has a much smaller range.

This is because the data transfer time taken between the server and the peer dominates all other factors. Here, the computational cost of a transaction is only about 10% of the total cost as compared to the LAN scenario where it is about 75%. Furthermore, the performance variations due to the different architectures become insignificant. The slight differences between the three architectures seen in the figure are likely due to changes in the network link characteristics during the course of running the experiments. The non-transactional transfer rate in this case corresponds to 2.7 transactions/sec which is within 7% of the lowest transactional rate of any of the architectures at any commit percentage.

The results for the second WAN link scenario (WAN2-SG). where the peer is located on a PlanetLab node in Singapore, are shown in Figure 6.14. Here the variation in the transactional rate due to the architectures or the commit rate is even less. In fact even with a y-axis with a very narrow range, hardly any variation is seen. The corresponding non-transactional rate in this case is 0.76 transactions per second, which

% Commit	TLA				2CA				SMA			
		Time per tx (s)				Time per tx (s)				Time per tx (s)		
	tx/sec	ave	min	max	tx/sec	ave	min	max	tx/sec	ave	min	max
0	0.75	1.34	1.14	2.33	0.75	1.33	1.32	2.34	0.75	1.33	1.32	2.35
16	0.75	1.34	1.30	2.32	0.75	1.33	1.30	2.19	0.75	1.34	1.30	2.34
41	0.75	1.33	1.30	2.33	0.75	1.33	1.30	2.32	0.75	1.33	1.30	2.33
57	0.76	1.32	1.30	2.32	0.76	1.32	1.30	2.19	0.76	1.32	1.30	2.33
84	0.76	1.32	1.30	2.32	0.76	1.32	1.30	2.32	0.76	1.32	1.30	2.34
100	0.76	1.32	1.30	2.20	0.76	1.31	1.30	2.32	0.76	1.31	1.30	2.32

Table 6.3: Average transaction rate for the three architectures in the WAN1–SG scenario. Also listed are the average, minimum, and maximum times taken for a transaction.

is the same as that for the three architectures when the commit percentage is greater than 57%. The results are summarized in Table 6.3.

6.6 Summary

In this work, we have proposed and compared the design and performance of three system architectures to support a transactional network interface. A transactional network interface allows an STM system to support network read/write operations within an atomic block. In addition to STM, there are several other potential uses of a transactional network interface. For example, the proposed architectures can be used to extend exception handling systems that rollback state and re-execute code to support network read/write operations. The logger implemented in TLA and 2CA can be used for building a fault-tolerant server where server applications may be non-deterministic in nature.

The performance results show that the overhead of using a transaction network interface is relatively insignificant. In fact, for connections with long RTT values, the overhead becomes negligible. Also, the performance approaches that of the non-transactional transfer rate as the number of aborts go to zero. In a well engineered system, it is highly likely that the abort rate is low during normal operation.

In terms of performance, no single architecture is clearly superior to the others.

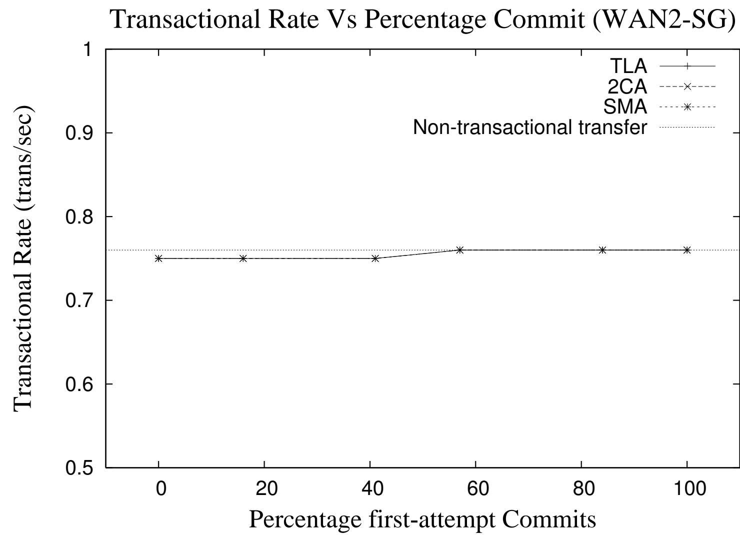


Figure 6.14: Transactional rate versus percentage of first-attempt commit transactions for the **WAN2-SG scenario**.

However, there are other design constraints that may make one architecture preferable to another in a particular situation. For example, unlike 2CA, TLA provides an end-to-end TCP connection and is efficient since it avoids kernel-to-user and user-to-kernel space copying. However, 2CA has the advantage that it does not require any OS changes on the logger. Similarly, TLA or 2CA is preferred over SMA if the user does not want a logger on each application server and would like to reuse an existing logger machine for that purpose.

Chapter 7

Efficient, Scalable Migration of IP Telephony Calls for Enhanced Fault-tolerance

IP telephony or VoIP has seen tremendous growth in the past few years and is predicted to grow rapidly in the coming years. Hence, IP telephony components must provide service with high-availability, comparable to traditional telephony systems. In this chapter, we present an efficient and scalable fault-tolerance mechanism for migrating calls to an alternate IP telephony call controller in the event of the failure of the call controller or network connectivity to it. We also present an efficient algorithm for merging components of migrated calls, so that the same call features are available on these calls as the original calls. Some of these techniques have been incorporated in commercial products, resulting in improved fault-tolerance.

7.1 Introduction

IP telephony or VoIP (Voice over IP) has seen tremendous growth in the past few years and is predicted to grow rapidly in the coming years. Both business and residential customers are switching to IP telephony, attracted not only by cost savings associated with convergence of data, voice and video, but also by the numerous compelling services and applications that arise from the integration of voice and Internet-based applications such as email and instant messaging. Furthermore, for businesses, IP telephony leads to simplified network administration and management due to convergence of voice and

data networks.

To be universally accepted, IP telephony must provide telephony services with the same degree of reliability and availability as circuit switched based telephony systems. A large body of research has focused on QoS and reliability of transporting voice over the Internet [46, 37], however, not much attention has been given to efficient fault tolerance techniques for other components of a VoIP system. Specifically, it is challenging to provide an efficient and scalable mechanism for call-preserving recovery of IP telephony components on failure of their call controller. In this chapter, we describe such a call-preserving mechanism for migrating calls to an alternate call controller. Furthermore, we present an efficient algorithm for merging call components of migrated calls.

In IP telephony systems, clients (such as IP endpoints and VoIP gateways e.g. Avaya G700 Media Gateway) usually connect to a call controller (such as Avaya Communication Manager) over an IP network for basic call services and various call features. Failure of the IP network or the call controller (CC) leads to service outage at the gateways and IP endpoints. This problem is usually addressed by installing alternate CCs that can provide service in the event of such a failure. However, when gateways or IP endpoints migrate to a new controller, existing calls on them may get torn down. Even if they are not torn down, no CC-implemented call features are available on these calls since the new controller has no knowledge of the state information associated with them. To be able to preserve not only the call bearer connections but also call features on such existing calls requires that the call state information be made available to the new controller. We refer to the ability of moving a call from one controller to another, while preserving the bearer connections and CC-implemented call features, as call migration. This ability is very useful in providing seamless service in not only failure situations but also during controller maintenance (e.g. upgrade of HW/SW on a CC) and potentially for load balancing, moving calls to a preferred controller, etc.

Call migration requires sharing or transferring of call session state information

between call controllers. We propose an efficient and scalable mechanism to share call session state for implementing call migration. This technique is similar in concept to the HTTP cookie mechanism [40]. The basic idea is that a CC saves call session state information on IP telephony entities such as VoIP gateways and IP endpoints; on failover, the alternate CC retrieves the call session state information from the failed-over entity and uses this information to reconstruct calls.

"Client side session cache" mechanism [66] is another conceptually similar technique. It can be used by a server to save TLS session state information on a client; this information can be used later to restart a TLS session, saving time since a new session does not have to be negotiated. The objective of saving the session information at the client, however, is not fault-tolerance, but to ease storage requirements at the server. The alternative – saving session state at the server – may be impractical in situations where the server interacts with a large number of clients.

The next section provides a functional overview of an IP telephony system. This is followed by goals for the fault-tolerance mechanism described in this work. Section 7.4 describes the call migration technique being proposed here. Merging components of a migrated call, spread over multiple VoIP gateways, is discussed in Section 7.5. Finally, we summarize this work in Section 7.6.

7.2 Architecture overview of a IP telephony system

A high-level view of an IP telephony system is shown in Figure 7.1. The figure identifies the main functional components of such a system; however, it does not show how such a system may be implemented. Indeed, in an actual instantiation of such a system many of the functional components may be distributed or implemented together or co-located. Furthermore, they may be geographically separated and connected by wide area network links. The main functional components, shown in the figure, are briefly discussed below.

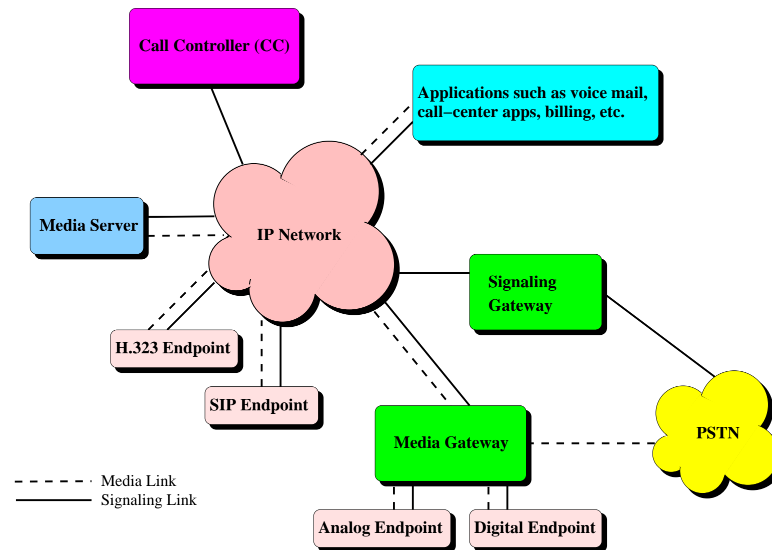


Figure 7.1: Functional architecture of an IP telephony system

Call Controller (CC). The CC functional component aggregates together a number of functionalities such as that of a H.248 [29] media gateway controller (MGC), H.323 [30] gatekeeper, feature server, SIP registrar and proxy, etc. It provides call logic and call control functions, and also typically maintains call state information.

An excellent description of the H.248 and H.323 protocols is provided in [75] and [41], respectively.

VoIP Gateways. Considering the huge amount of existing investment in traditional TDM-based telephony, it is evident that IP and TDM-based telephony systems will coexist and will need to inter-operate for a long time to come. Thus, an important part of the system is gateways which allow communication between IP and TDM-based circuit switched systems. A media gateway (MG) terminates media streams from more than one kind of network, e.g. IP and TDM, and thus allows those different networks to inter-operate. Similarly, a signaling gateway (SG) converts signaling protocols from one kind of network to another. Specifically, MGs and SGs are essential for inter-working between IP and TDM based public switched telephone network (PSTN). For example,

if an analog or digital phone needs to call an IP phone, both voice and signaling for the call will pass through such gateways. In this chapter, we use the terms gateway and VoIP gateway interchangeably.

Media Server. A media server provides a number of media services such as announcements (e.g. **the number you have dialed has been disconnected...**), music on hold, etc. It also provides DTMF tone detection (which is, e.g, used for recognizing tones of the digits pressed while making an automated credit card payment on the phone), and, call classification (which involves appropriately routing a call based on caller input, including using automatic speech recognition techniques). A media server may also provide resources for hosting conference calls.

Endpoints. Endpoints are the devices used for making and receiving calls. These can be traditional (that is, analog or digital, e.g., Avaya DCP phones) or IP endpoints. An IP endpoint typically uses SIP or H.323 protocols for call signaling. It is usually also available as a softphone, an IP endpoint application that runs on a generic computer. In this chapter, the term endpoint without qualification refers to an IP endpoint.

7.3 Requirements/Goals

We were guided by the following goals while designing this failure-recovery mechanism.

- **Failover to an alternate CC.** Gateways and endpoints failover to an alternate CC in the event of failure of their current CC or failure of network connectivity to it.

- **Call session preservation.** Call session state of stable calls is preserved on failover of gateways and endpoints to an alternate CC. A call is considered stable if it is fully established, that is, all signaling messages required to setup

the call have been exchanged and the voice paths already established. Also, we distinguish between preserving just the connection, i.e. the voice path, of a call (connection preservation) and preserving the connection as well as the call session state of the call (call session preservation). In case of connection preservation, it may not be possible to make changes to the state of the call. Specifically, call features implemented at the CC are not accessible to the call, e.g. the user may not be able to conference in another party to the call. On the other hand, in case of call session preservation, the call session state is available at the alternate CC which allows the users on the call to continue to access call features offered by the CC.

- **Seamless failover and recovery.** Failover to an alternate CC and recovery are seamless from the users' point of view, that is, they do not experience an outage.

- **Scalability.** The fault-tolerance mechanism scales well, preferably linearly, with increase in the number of gateways and endpoints.

- **Efficiency and minimal overhead.** The fault-tolerance mechanism is efficient such that there is minimal overhead during normal operation (no failures). Furthermore, failover and recovery is fast so that there is minimal disruption in the continuity of service to the user.

7.4 Call Migration

Without loss of generality, we will consider the IP telephony system elements shown in Figure 7.2 for discussing call migration. During normal operation (failure-free periods), the gateways and endpoints receive service from the main call controller. The VoIP gateways (VGs), shown in the figure, combine the functionality of both media and

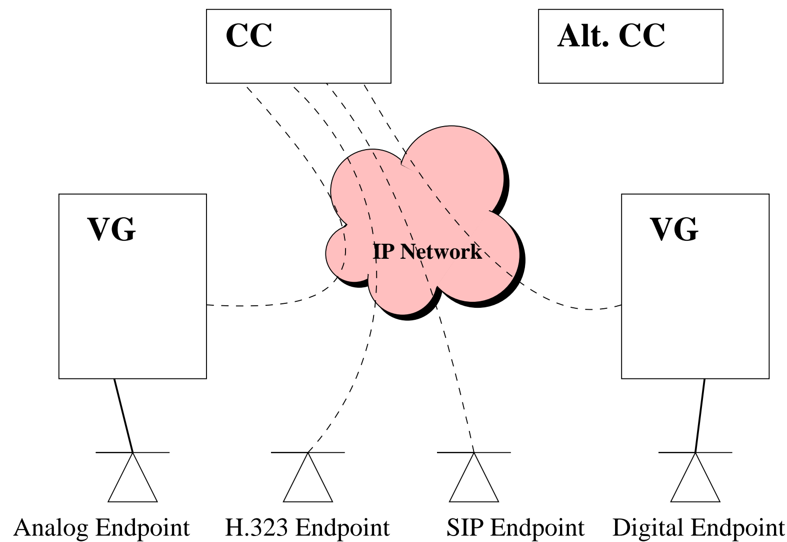


Figure 7.2: An IP telephony system.

signaling gateways. In addition, they also act as media servers. Depending on the needs of the organization deploying the IP telephony system, the VGs may be distributed over a wide area and connect to the main CC over WAN links.

If the main CC, or network connectivity to it, fails, the gateways and endpoints seek service from an alternate CC. Based on the degree of service availability required, there may be one or more alternate CCs. Usually, VGs and endpoints that are remotely located are provided with a local alternate CC for failure scenarios where the remote location may lose connectivity with the main location.

In order to provide seamless service, calls on a particular VG are migrated to an alternate CC when that VG fails over. To accomplish this migration, the alternate CC needs state information related to the existing calls. There are various ways to synchronize this information to alternate CC(s). However, the mechanism we describe below is more scalable, robust and efficient as compared to others. We compare our methodology with the alternatives in Section 7.4.3.

Note that call migration can also be used in non-failure conditions for moving

gateways and IP endpoints, and, migrating the associated calls from one controller to another for reasons other than failures such as maintenance, load balancing, etc.

7.4.1 Saving State on Clients

As calls are made, a CC saves the call state information on the MGs and IP endpoints, similar to how an HTTP server saves client state information on the client machines as HTTP cookies [40]. When a CC fully establishes a call, it generates one or more messages with call state information and sends them to the VG and/or IP endpoint(s) involved in that call. If the state of a call changes (e.g. another party is added to the call), the controller sends out updated message(s). When a call ends, the CC sends out a delete message to purge the call state information related to that call. For performance efficiency, the cookie messages can be piggy-backed on regular signaling messages exchanged between the CC and VGs/endpoints.

Like an HTTP cookie, the call state information is opaque to the client, that is, a VG or an IP endpoint has no knowledge of its content; only call controllers are aware of its structure and semantic content. This is very advantageous as changes can be made to the structure or content of a call state information message without requiring any code changes at all on the VGs or IP endpoints.

7.4.2 Reconstructing Calls

When a VG or an IP endpoint fails over to an alternate call controller, as part of initial negotiation, all the existing calls on the VG or IP endpoint are migrated to the new call controller. This new CC queries [1] the VG or IP endpoint for its call state information. This state information is then used to reconstruct (as far as possible) the existing calls by recreating data structures and state that represented the calls in the original controller. This allows the new controller to provide call features on those calls.

[1] For a H.248 gateway, the H.248 audit mechanism [29] can be used for this purpose.

7.4.3 Comparison to alternatives

Beside saving the call session state at the clients (gateways and IP endpoints) of the CC, there are other ways of synchronizing the state information between the CCs. We compare our approach with two such approaches.

- The call controller and the alternate call controllers can use a common, redundant backend database where all pertinent call state information can be stored. This allows all the call controllers to have access to the same call state data. This approach has the following shortcomings.

  * Such a common, backend database can be expensive to build and maintain, considering that call session state data is very dynamic and voluminous (up to hundreds of thousands of calls per hour)
  * The database can become a bottleneck, especially as the number of controllers accessing the database increase.
  * This solution is not very robust if the gateways are geographically distributed as they often are for big enterprises, since WAN network failures can prevent alternate controllers from accessing the common database. For example, consider an enterprise IP telephony system with a main and multiple remote locations. During normal operation all the remote locations are served by the CC and the redundant database at the main location; all the remote locations have alternate CCs in case they lose connectivity to the main location. Here, if the WAN link to the main location were to fail, the alternate CC at this location will not be able to access the database.

- The call controller dynamically updates the call session state information on the alternate call controllers. Conceptually, this is similar to having a database

at the main location which is replicated in real time at all the alternate CC locations. This approach has a number of drawbacks.

* It adds a lot of overhead and complexity as the number of call controllers increase.
* The alternate controllers have to be updated with the call state information of all the calls on the main controller. This can lead to network congestion if the link to a particular alternate controller does not have sufficient bandwidth.
* This solution is not very robust, since network failures can prevent controllers from communicating with each other.

7.4.4 Advantages

We now summarize the main advantages of our approach.

(1) **Enhanced Reliability.** Our solution is highly robust to network failures due to its highly distributed nature.

(2) **Enhanced Scalability.** Since VGs and IP endpoints carry their own call state information, this solution easily scales to any number of alternate call controllers. In contrast to the two approaches described above, increase in the number of alternate call controllers does not require additional processing, nor does it introduce any bottlenecks.

(3) **Efficiency.** Only state information that is needed is transferred to an alternate controller and it is done only when needed.

(4) **Simplicity.** Requires less resources and is simpler to implement than other approaches.

7.5 Merging call components during reconstruction

As described earlier, when a VoIP gateway migrates to an alternate call controller, it queries the gateway for call session state information (saved by the call controller that originally created the call) in order to reconstruct the calls on that gateway.

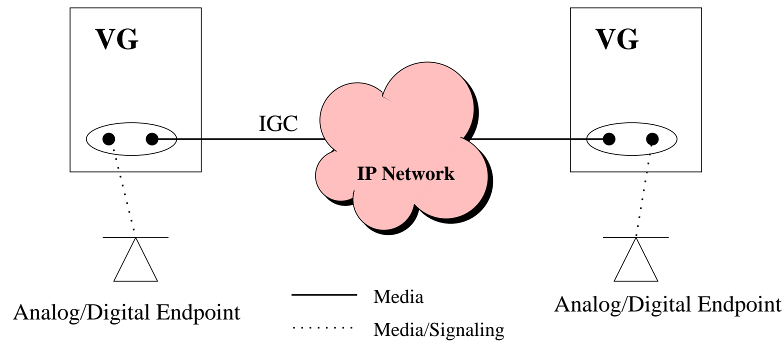


Figure 7.3: An inter-gateway connection.

A call may span more than one gateway. For instance, if an analog/digital endpoint on one VG calls another on another VG, an IP bearer path is created between the two VGs to setup the call. Such an IP bearer connection between two VGs is called an inter-gateway connection (IGC) and connects two parts of a call as shown in Figure 7.3. A conference call may involve multiple VGs and contain multiple IGCs.

Since VGs (and endpoints) may detect failure at different times, a call spanning multiple VGs will not failover in its entirety at one instance. Rather, it will failover in parts as the VGs and endpoints containing those parts failover.

As a CC processes each entity that fails over, it reconstructs the calls residing on that entity. Calls that span multiple entities have to be treated specially, since when a part of the call is being processed, the CC has to determine whether to reconstruct it into a new call or merge it into an existing call. We present an algorithm here that, during call reconstruction, correctly and efficiently merges call components on different entities, which migrate to a controller, into the same call.

7.5.1 Advantages of merging a call

The simplest solution to merging a call is, in fact, to leave the components of a call unmerged, that is, let each component of the call be reconstructed as a separate call. This preserves the talk path, however, some of the call features will not work correctly since the structure of the call is different from the original. In other words, it is connection preserving, but may not preserve the call session faithfully.

For example, it is possible that a component reconstructed separately as a call has no disconnect supervision (ability to signal a disconnect). In Figure 7.4, we show a call between a phone (P1) and a trunk (T1). We assume that the trunk does not have disconnect supervision. When P1 hangs up, the entire call is torn down. However, this is not the case if the VGs migrate to another CC and the two components of the call are reconstructed as separate calls. The call with the trunk termination, Call 1 in Figure 7.4 (b), does not have disconnection supervision. Note that the alternate CC does not the know the identity of the remote ends of IGCs in both Calls 1 and 2.

Thus, to summarize, the main advantage of merging the components of a call is that the reconstructed call would behave exactly the same with respect to all the call features as the original call.

7.5.2 Call components merging algorithm

One strategy to merging components of a call is to assign an identifier to uniquely identify each call. In addition to be unique for each call generated by a CC, this identifier would also have to be unique across different CCs and across reboots of the same CC. In our algorithm, presented below, we do not use such a unique identifier and thus save the resources required in its generation and management.

Instead, to identify components of a call, at each IGC endpoint the identity of the remote end of that IGC is saved. This identity consists of two pieces of information

which are saved by the CC at the gateway when a call is created:

(1) Far end gateway ID (FEGI)

(2) Far end context ID (FECI) [2]

(Here, we assume that the VGs are controlled by the CC using H.248 protocol. However, the merging algorithm is applicable to any VoIP gateway.)

This information is saved as a property of the ephemeral terminations involved in the IGC. During reconstruction on the alternate CC this information is retrieved from the VG and is used to merge IGCs. Whenever a call containing an IGC is to be reconstructed, it is first checked if the IGC in the incoming call matches an IGC in an already reconstructed call. If there is a match, the incoming call is merged with the matched call, else, it is reconstructed into a new call. The IGC merging algorithm can be described through the following two basic scenarios:

- **Scenario 1: Call involving one IGC.** This scenario is depicted in Figure 7.5. This call has two parts – C1 on VG1 and C2 on VG2. Here, on failover, the alt. CC can uniquely match the two parts of the call as the FEGI and the FECI of the two ends of the IGC point to each other.

- **Scenario 2: Call involving multiple IGCs.** This scenario is described in terms of the example in Figure 7.6, where three VGs are involved in a conference call which is hosted on VG1. There are two IGCs. Unlike in the previous scenario, here the order in which the VGs migrate to the alt. CC is significant. Two cases are possible depending on the order of VG failover (all other cases can be reduced to these two, even when more than two IGCs are involved):

  * **Case 1:** VG2 first migrates to the alternate CC, followed by VG3. Here, parts of the call (C2 and C3) can be uniquely matched as FEGI and FECI

[2] This is the H.248 context ID. However, any other gateway local ID that represents an association between the terminations in a connection can also be used.

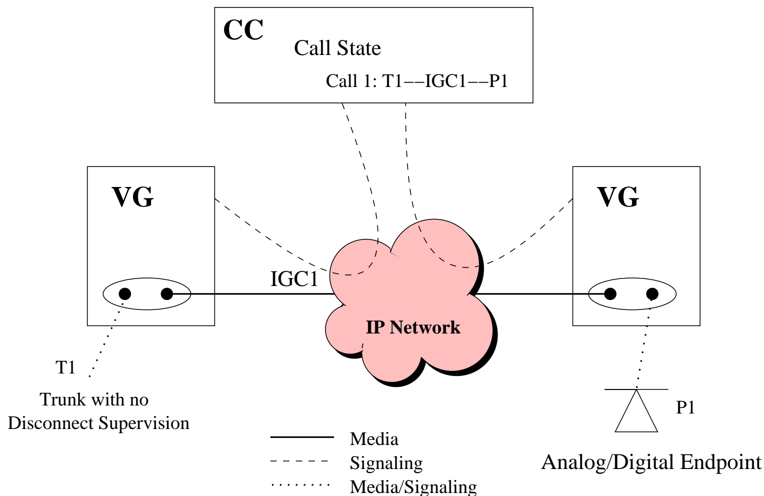
of the two ends of the IGC point to the same gateway (VG1, here) and context (C1, here) respectively.

* **Case 2:** VG2 first migrates to the alternate CC, followed by VG1. This case is identical to the scenario 1 above.

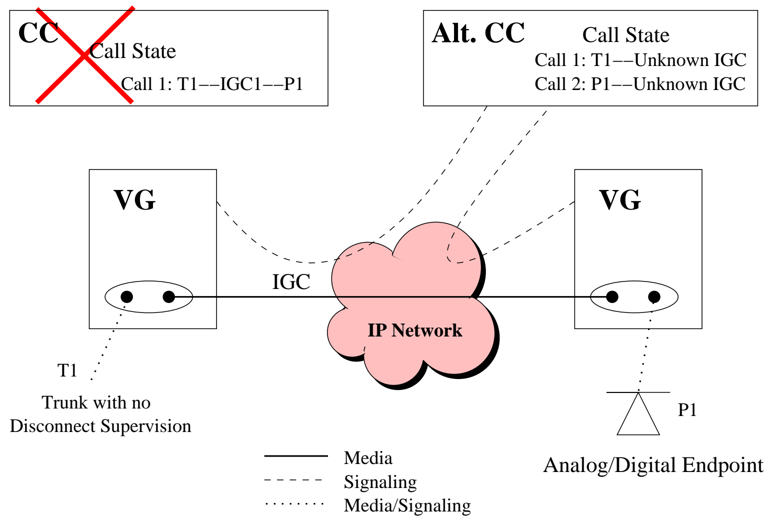
7.6 Summary

IP systems must provide toll-quality telephony services and maintain continuity of service. In this chapter, we presented an efficient and scalable fault-tolerance mechanism to migrate calls from one CC to another. CCs save call session state on the IP telephony entities that host the call. This allows these entities to carry call session state information with them and seek service from an alternate CC during failures. Since the entities supply call state information, the alternate CC is able to provide call features for the session created on the original CC. The alternate CC uses an efficient algorithm to merge components of calls so that the calls retain their original structure when reconstructed.

Some of these mechanisms have been implemented in Avaya IP telephony products resulting in preserved calls and better user experience during failures.



(a) Before failure



(b) After failover

Figure 7.4: (a) A call between a trunk, T1 and a phone, P1 is created. (b) After failure of the CC, the VGs migrate to the alternate CC. The original call gets reconstructed as two different calls.

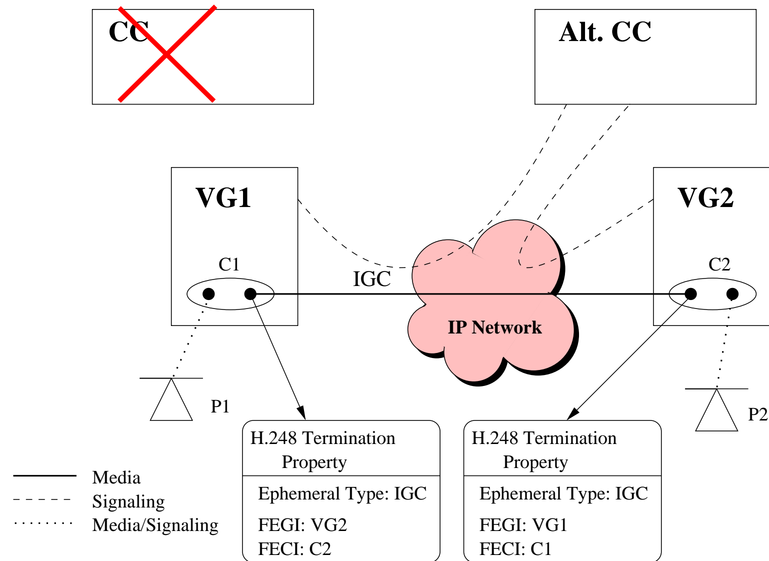


Figure 7.5: A call between P1 and P2 is shown. After failure of the CC, VG1 and VG2 migrate to the alt. CC, which uses the H.248 properties saved at the IGC terminations to merge the two ends of the call.

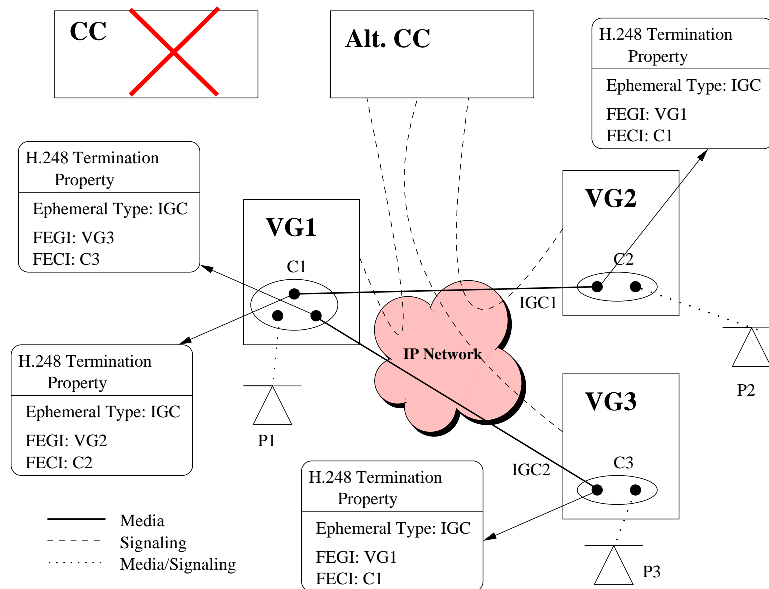


Figure 7.6: This shows a three party call. After the failure of the CC, the order in which the VGs migrate to the alt. CC is significant. The alt. CC uses the merge algorithm described here to merge the three parts of the call.

Chapter 8

Conclusions and Future Work

In this dissertation, we present three systems that provide enhanced server fault-tolerance with the aim to improve user experience during server failures. This requires session-preserving, fast and seamless failovers. The three systems are: ST-TCP, CRAFT and call preserving migration of IP telephony calls.

ST-TCP is very efficient, with a fast failover, and an insignificant overhead during normal operation. It uses an active backup with a replica application. ST-TCP does not require any changes to server applications, however, it requires that they be deterministic.

To address some of the drawbacks of ST-TCP, mainly a tightly coupled backup and inability to handle non-determinism, we designed CRAFT. CRAFT uses TCP splicing and resplicing at a proxy for transport layer mobility. For application layer failover, the requests and responses are logged, transactionalized and tagged. Transactionalization allows replay of the failed requests, while tagging facilitates intelligent replay. The changes required in the application are minimal and an adaptive failure detection mechanism is used. We deployed a prototype implementation of our architecture with an open-source webmail application called RoundCube Webmail and the results showed that the entire failover process takes only a few seconds even for clients connected over a WAN link. This is without any expensive client-server heartbeats.

The final part of the thesis describes a fault-tolerance mechanism for a highly

distributed, IP telephony system. The key issue – of synchronization of a large amount of centralized, dynamic call state information – is addressed by saving call session state information at endpoints. This allows efficient, scalable and seamless migration of IP telephony calls to alternate call servers on a call server failure. Table 8.1 provides a comparison of the three systems.

	ST-TCP	CRAFT	IP Call Server FT
Failover time	$\approx$ 1s	$\approx$ 1s (proxy) $\approx$ 3s (backend server)	A few minutes (but talk path is preserved)
Session preserving	Yes	Yes	Yes
Changes to server App.	No	Yes	Yes
Changes to server OS	Yes	No	No
Changes to client	No	No	Yes (but minimal)
Dedicated backup	Yes	No	No
Deterministic server App.	Yes	No	No

Table 8.1: Comparison between the three server fault-tolerance mechanisms.

Some future research directions that explore issues not addressed in the current work are presented below.

- **Application level support.** CRAFT requires some application level support (Section 5.3.3). An interesting and challenging problem is to implement CRAFT with no application level modifications required. However, if that is not possible, development of tools that assist in making any required application level modifications will be useful.

- **OS Virtualization.** Considering the rapid growth of OS virtualization technologies such as Xen and VMware, it will be interesting to explore how CRAFT could be used in a virtualized environment. Note that although it is easy to migrate virtual machines (VMs) during normal operation, seamlessly migrating a VM on server failure is a hard problem.

- **Non-determinism.** Although CRAFT addresses non-determinism (Section

5.3.4), automated tools that assist in detection of non-deterministic transactions for a particular application will be very useful.

- **Geographical Redundancy.** For providing business continuity and seamless service during disaster recovery, it is necessary that the primary and an alternate server be geographically separated. In our architectures, the backup (in case of ST-TCP) and an alternate server (in case of CRAFT) are assumed to be on the same subnet as the primary server. This limits their geographical separation. Removing this limitation will be a significant research contribution.

- **Multiple Backups.** In both ST-TCP and CRAFT, only a single backup server is considered. These architectures can be extended to support multiple backups, which would handle multiple server failure scenarios.

- **Security Implications.** In this dissertation, the security issues related to our systems architectures are not addressed. An important future research effort could be to explore these issues.

- **Scalability.** Although our prototype implementation of CRAFT provides a proof of concept, we have not addressed all the scalability issues pertaining to backend server fault-tolerance described in Chapter 5. Specifically, testing the system with hundreds to thousands of distributed clients will provide important data points for further tuning the system architecture. Furthermore, in addition to `Roundcubemail`, other applications should be used in conjunction with CRAFT.

Bibliography

[1] Navid Aghdaie and Yuval Tamir. Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN 2002), Miami, FL, Oct. 2002.

[2] Ajax, http://wikipedia.org/ajax.

[3] Alteon websystems, http://www.alteonwebsystems.com.

[4] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side tcp to mask connection failures. In Proceedings of Infocom 2001, April 2001.

[5] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. ACM Transactions on Computing Systems, 13(4):311–342, 1995.

[6] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS-98-4, 1998. Available at: http://www.cnds.jhu.edu/publications/.

[7] Apache. http://apache.org.

[8] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In Proceedings of USENIX'99, 1999.

[9] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In Proceedings of USENIX Annual Technical Conference, General Track, 2000.

[10] ARP Issue, http://www.linuxvirtualserver.org/docs/arp.html.

[11] James Aweya, Michel Ouellette, Delfin Y. Montuno, Bernard Doray, and Kent Felske. An adaptive load balancing scheme for web servers. Int. Journal of Network Management, 12(1):3–39, 2002.

[12] F. Baker. Requirements for IP version 4 routers. Request For Comments 1812, Internet Engineering Task Force, 1995.

[13] P. Barret, A. Hilborne, P. Bond, P. Verssimo D. Seaton, L. Rodrigues, and N. Speirs. The delta-4 extra performance architecture (xpa). In In Proc. of 20th IEEE Symp. on Fault-Tolerant Systems (FTCS-20), pages 481–488, 1990.

[14] Andy C. Bavier, Mic Bowman, Brent N. Chun, David E. Culler, Scott Karlin, Steve Muir, Larry L. Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating systems support for planetary-scale network services. In NSDI, pages 253–266. USENIX, 2004.

[15] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems, 9(3):272–314, Aug 1991.

[16] 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.

[17] N. Budhiraja and K. Marzullo. Highly-available services using the primary-backup approach. In Proceedings of the 2nd Workshop on Management of Replicated Data, Monterey, CA, 1992.

[18] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. ACM Comput. Surv., 34(2):263–311, 2002.

[19] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic load balancing on web-server systems. IEEE Internet Computing, 3(3):28–39, 1999.

[20] Common internet file system, http://samba.org/cifs.

[21] Cisco content services switches, http://www.cisco.com.

[22] Ariel Cohen, Sampath Rangarajan, and J. Hamilton Slye. On the performance of tcp splicing for url-aware redirection. In USENIX Symposium on Internet Technologies and Systems, 1999.

[23] M. Crispin. Internet Message Access Protocol - Version 4, Rev1. Request For Comments 2060, Internet Engineering Task Force, 1996.

[24] F. Cristian. Understanding fault-tolerant distributed systems. Communications of ACM, 34(2):56–78, Feb 1991.

[25] Ethereal, http://ethereal.com.

[26] Christof Fetzer. Enforcing perfect failure detection. IEEE Transactions of Computers, Feb. 2003.

[27] Yahoo's flickr, http://flickr.com.

[28] Gmail Blog, http://gmailblog.blogspot.com/2007/10/code-changes-to-prepare-gmail-for.html.

[29] ITU-T Recommendation. H.248: Gateway control protocol, June 2000.

[30] ITU-T Recommendation. H.323: Packet based multimedia communications systems, Feb. 1998.

[31] Tim Harris. Exceptions and side-effects in atomic blocks. Sci. Comput. Program., 58(3):325–343, 2005.

[32] Eric Van Hensbergen and Athanasios E. Papathanasiou. KNITS: Switch-based connection hand-off. In IEEE Infocom, 2002.

[33] Special notes for HA HP CIFS server, http://docs.hp.com/en/b8725-90053/ch06s05.html.

[34] Pai-Hsiang Hsiao, H. T. Kung, and Koan-Sin Tan. Video over tcp with receiver-based delay control. In International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001), pages 199–208, June 2001.

[35] H. Y. Huang and C. Kintala. Software implemented fault tolerance. In Proceedings of the IEEE Fault Tolerant Computing Symposium, Toulouse, France, June 1993.

[36] Ibm interactive network dispatcher, http://www.ibm.com/dispatcher.

[37] Mark Karol, P. Krishnan, and J. Jenny Li. VoIP network failure detection and user notification. In Proceedings of the IEEE Int. Conf. on Computer Communications and Networks, Oct. 2003.

[38] Rupert R. Koch, Sanjay Hortikar, Louise E. Moser, and P. Michael Melliar-Smith. Transparent TCP connection failover. In Proceedings of the IEEE Int. Conf. on Dependable Systems and Networks, San Francisco, June 2003.

[39] Ravi Kokku, Ramakrishnan Rajamony, Lorenzo Alvisi, and Harrick Vin. Half-pipe anchoring: An efficient technique for multiple connection handoff. In Proceedings of ICNP, Paris, France, Nov 2002.

[40] D. Kristol and L. Montulli. HTTP state management mechanism. Request For Comments 2109, Internet Engineering Task Force, 1997.

[41] Hong Liu and Petros Mouchtaris. Voice over IP signaling: H.323 and beyond. IEEE Communications Magazine, Oct. 2000.

[42] LVS project, http://www.linuxvirtualserver.org.

[43] D. Maltz and P. Bhagwat. TCP splicing for application layer proxy performance, 1998.

[44] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In Proceedings of INFOCOMM'98, March 1998.

[45] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.

[46] A. Markopoulou, F. Tobagi, and M. Karam. Assessment of VoIP quality over internet backbones. In Proceedings of IEEE Infocom, June 2002.

[47] 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.

[48] M. Marwah, S. Mishra, and C. Fetzer. A system demonstration of ST-TCP. In Proceedings of IEEE Int. Conf. on Dependable Systems and Networks, Yokohama, Japan, June 2005.

[49] Manish Marwah, Shivakant Mishra, and Christof Fetzer. Fault-tolerant and scalable tcp splice and web server architecture. In SRDS, pages 301–310. IEEE Computer Society, 2006.

[50] Manish Marwah, Shivakant Mishra, and Christof Fetzer. Systems architectures for transactional network interface. In 10th IEEE High Assurance Systems Engineering Symposium, Dallas, TX, Nov. 2007.

[51] S. Mishra, C. Fetzer, and F. Cristian. The Timewheel group communication system. IEEE Transaction on Computers, 51(8), August 2002.

[52] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Distributed Systems Engineering, 1(2):87–103, Dec 1993.

[53] Netfilter, http://www.netfilter.org.

[54] D. Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In 4th USENIX Symposium on Internet Technologies and Systems (USITS '03), March 2003.

[55] M. Orgiyan and C. Fetzer. Tapping TCP streams. In Proceedings of the IEEE International Symposium on Network Computing and Applications, February 2002.

[56] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-aware request distribution in cluster-based network servers. In ASPLOS, pages 205–216, 1998.

[57] David A. Patterson. A simple way to estimate the cost of downtime. In Proceedings of LISA '02: Sixteenth Systems Administration Conference, November 2002.

[58] Jon Postel. Transmission control protocol. Request For Comments 793, Internet Engineering Task Force, 1981.

[59] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design. ACM Computing Surveys, 10(2):123–166, Jun 1978.

[60] A. Rijsinghani. Computation of the Internet Checksum via Incremental Update. Request For Comments 1624, Internet Engineering Task Force, May 1994.

[61] Marcel-Catalin Rosu and Daniela Rosu. An evaluation of TCP splice benefits in web proxy servers. In WWW, pages 13–24, 2002.

[62] Marcel-Catalin Rosu and Daniela Rosu. Kernel support for faster web proxies. In USENIX Annual Technical Conference, General Track, pages 225–238, 2003.

[63] Roundcube webmail, http://roundcube.net.

[64] Samba. http://samba.org.

[65] R. Sandberg. Sun network filesystem protocol specification. Technical report, Sun Microsystems, Inc., 1985.

[66] H. Shacham and D. Boneh. Fast-track session establishment for TLS. In Proceedings of the Network and Distributed System Security Symposium, 2002.

[67] Nir Shavit and Dan Touitou. Software transactional memory. In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 204–213. Aug 1995.

[68] G. Shenoy, S.K. Satapati, and R. Bettati. HYDRANET-FT: Network support for dependable services. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, Taipei, Taiwan, April 2000.

[69] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, March 2001.

[70] Oliver Spatscheck, Jørgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP forwarder performance. IEEE/ACM Transactions on Networking, 8(2):146–157, 2000.

[71] R. Stewart and et. al. Stream control transport protocol. Request For Comments 2960, Internet Engineering Task Force, 2000.

[72] 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.

[73] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Connection migration for service continuity over the internet. In Proceedings of the 22th IEEE International Conference on Distributed Computing Systems, Vienna, Austria, July 2002.

[74] W. Tang, L. Cherkasova, L. Russell, and M. Mutka. Modular tcp handoff design in streams-based tcp/ip implementation. In Lecture Notes in Computer Science, volume 2094. Springer-Verlag, 2001.

[75] Tom Taylor. Megaco/H.248: A new standard for media gateway control. IEEE Communications Magazine, Oct. 2000.

[76] TCP-RTM: Using TCP for real time applications, http://www-dsg.stanford.edu/sliang/rtm.pdf.

[77] Linux TCP splicing module, http://www.linuxvirtualserver.org/software/tcpsp.

[78] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. Communication of the ACM, 39(4):76–83, Apr 1996.

[79] T. Wilson. The cost of downtime. Internetweek.com, July 1999.

[80] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. 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.

[81] Li Zhao, Yan Luo, Laxmi Bhuyan, and Ravi Iyer. Design and implementation of a content-aware switch using an network processor. In 13th HOT Interconnect, 2005.

[82] H. Zou and F. Jahanian. Real-time primary-backup (RTPB) replication with temporal consistency guarantees. In Proceedings of the 18th International Conference on Distributed Computing Systems, 1998.