

UNIVERSITY OF CALIFORNIA

Los Angeles

Transparent Fault-Tolerant Network Services Using Off-the-Shelf Components

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Navid Aghdaie

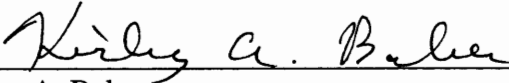
2005

© Copyright by

Navid Aghdaie

2005

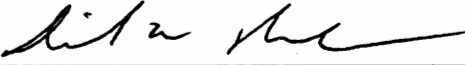
The dissertation of Navid Aghdaie is approved.



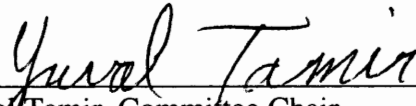
Kirby A. Baker



Mario Gerla



David A. Rennels



Yuva Tamir, Committee Chair

University of California, Los Angeles

2005

To my parents and sister.

Table of Contents

Chapter One - Introduction	1
1.1. Network Services	3
1.2. Service Availability and Reliability	7
1.3. Client-Transparent Fault-Tolerance	9
1.4. Organization of this Work	12
Chapter Two - Related Work	14
2.1. Availability Solutions	15
2.1.1. DNS Based Load Balancing	15
2.1.2. Centralized Routers and Directors	16
2.1.3. Application-Level Redirect	19
2.2. Availability and Reliability Solutions	20
2.2.1. Client-Aware Solutions	21
2.2.2. Client-Transparent Server Fault Tolerance	25
2.3. Replicated Back-end Servers	31
2.4. Fault-Tolerant Video Conferencing	32
2.5. Summary	34
Chapter Three - Constructing Client-Transparent Reliable Network Services	36
3.1. Requirements for Adding Client-Transparent Fault Tolerance to Network Service	36

3.2. Fault Model and Assumptions	38
3.3. Service State	40
3.3.1. Service Identity	40
3.3.2. Connection State	41
3.3.3. Application State	42
3.4. Approaches for State Preservation	44
3.4.1. Active Replication	44
3.4.2. Message Logging	45
3.4.3. Checkpointing	46
3.5. Techniques for Handling Non-determinism	47
3.6. Error Detection	48
3.7. Service Failover	50
3.8. Restoration of Fault-Tolerant Service After Failover	51
3.9. Summary	52

Chapter Four - CoRAL: A Transparent Fault-Tolerant Web

Service	55
4.1. Assumptions	56
4.2. System Architecture	57
4.4. Implementation	61
4.4.1. A Proxy-Based User-Level Implementation	63
4.4.2. An Implementation Based on Kernel-Level and Web Server Modifications	69
4.4.2.1. Kernel Modifications	71

4.4.2.2. Modifications to the Server Application	74
4.4.2.2.1. Worker Processes	74
4.4.2.2.2. Mux/Demux Processes	76
4.4.3. Converting Process Crashes to Host Crashes	78
4.4.4. Failover	80
4.4.5. Restoration of Fault-Tolerant Service After Failover	83
4.5. Optimizations	85
4.5.1. Dual-Role Server Hosts	86
4.5.2. Efficient Handling of Static, Deterministic Content	88
4.6. Performance Evaluation	91
4.6.1. Experiment Setup	92
4.6.2. Failure-Free Performance	94
4.6.2.1. Application Reply Types and System Behavior	
.....	95
4.6.2.2. Dynamically Generated Replies	96
4.6.2.2.1. Latency Overhead with Dynamic Replies	
.....	98
4.6.2.2.2. Processing Overhead with Dynamic	
Replies	98
4.6.2.2.3. Throughput with Dynamic Replies	101
4.6.2.2.4. Evaluation of the Dual-Role Servers	
Optimization	103
4.6.2.3. Replies Based on Static Content	104

4.6.2.3.1. Latency Overhead for Static Content	105
4.6.2.3.2. Throughput with Static Content	106
4.6.2.3.3. Processing Overhead with Static Content	109
4.6.2.3.4. Optimization for Deterministic or Static Content	110
4.6.3. Service Failover and Recovery	114
4.6.4. Implementation Comparison: User-Level versus Kernel Support	119
4.6.5. Impact of Processor and Network Speed on System Performance	124
4.6.5.1. Latency	125
4.6.5.2. Throughput	126
4.6.6. Performance Under Overload	126
4.7. Fault Tolerance Validation Using Fault Injection	129
4.7.1. Register Fault Injections	131
4.7.2. Memory Fault Injection	133
4.8. Summary	135

Chapter Five - Transparent Fault-Tolerant Video Conferencing

.....	137
5.1. Introduction to Off-the-Shelf Video Conferencing	138
5.2. Adding Fault Tolerance to a Video Conferencing Server	139

5.2.1. Unreliable Communication	141
5.2.2. Application State and Non-determinism	145
5.3. Performance Evaluation	146
5.3.1. Processing Overhead	147
5.3.2. Failover Latency	149
5.3.3. Impact of Heartbeat Rate and UDP Configuration Choice	151
5.3.4. TCP Replication and Application Synchronization	152
5.4. Fault Tolerance Validation Using Fault Injection	153
5.5. Summary	156
Chapter Six - Summary and Conclusions	157
Bibliography	160

List of Figures

1.1 Typical Organization of Network Service Elements	4
1.2 Three-tier Web Service Architecture	5
1.3 Multi-tier Service Architecture	6
4.1 Message paths for a standard and a replicated server	58
4.2 CoRAL duplex system structure	66
4.3 CoRAL implementation with kernel and web server modules	70
4.4 Server module structure	75
4.5 CoRAL Failover	81
4.6 CPU cycles used for processing dynamic requests	87
4.7 Average latency for dynamic replies	97
4.8 Used CPU cycles for dynamic replies	99
4.9 System throughput for dynamic replies	101
4.10 Peak system throughput with dual-role optimization	104
4.11 Average latency for static replies	105
4.12 System throughput with static replies	108
4.13 CPU cycle overhead: full duplex versus optimization for static content	110
4.14 Reply latency with static and sync-static optimizations	112
4.15 Latency distribution	112
4.16 Peak system throughput with static and sync-static optimizations	

.....	114
4.17 System throughput during a fault	115
4.18 System unavailability due to a fault	116
4.19 Comparison of client measured request times for user-level implementation	120
4.20 Breakdown of duplex mode request times	121
4.21 Duplex mode response times for different HTTP message sizes	122
4.22 Standard server response times for different HTTP message sizes	122
4.23 Latency on Fast Hosts	125
4.24 Throughput on Fast Hosts	127
4.25 Performance degradation due to overload	128
5.1 Video Conferencing with a Multi-Conferencing Unit	138
5.2 Direct UDP Communication Between Client and Primary	142
5.3 UDP communication Using IP Multicast or A Multicast Node	144
5.4 CPU Cycle Overhead	148
5.5 Video Conferencing Failover Interruption Time	150
5.6 CPU Overhead for Different UDP Configuration	151

List of Tables

4.1 Recovery from server failures	61
4.2 Recovery from communication errors	62
4.3 Breakdown of used CPU cycles	109
4.4 Breakdown of failover latency	117
4.5 Breakdown of server integration latency	118
4.6 Average and median request times for user-level implementation	120
4.7 Breakdown of average and median response times for duplex mode	121
4.8 Average and median request times for different HTTP message sizes	123
4.9 Comparison of user-level and kernel implementations	124
4.10 CoRAL Register Fault Injection	132
4.11 CoRAL Memory Fault Injection	134
5.1 Video Conferencing Register Fault Injection	154

ACKNOWLEDGMENTS

I would not have reached my goals without the help and support of many people. I wish to express my gratitude to all of them.

I would like to especially thank my advisor, Professor Yuval Tamir. His vast knowledge and dedication to research helped me greatly in improving the quality of my research, presentations, and writing skills. I know my career will benefit greatly from the knowledge that I have gained from him. I am grateful for him guiding me through the correct path.

I enjoyed the discussions and group meetings with my colleagues at the UCLA Concurrent System Lab, Ming Li, Israel Hsu, Wenchao Yang, Edward Young, Daniel Goldberg, Yiguo Wu, Michael Le, Donald Lam, Benjamin Liao, Kahmyong Moon, and Yoshio Turner. I would like to thank them for their camaraderie. Their work has greatly benefited my research and knowledge. I would also like to thank Verra Morgan for her help and administrative support at the Computer Science Department's Graduate Student Affairs Office.

I was lucky to have gained some industry experience during this process. I learned a great deal about industry and the Web during my time at LiquidMarket / NBCi. I would like to especially thank Dr. Francois Rouaix who served as a great mentor, Gauthier Groult who gave me the opportunity, and my friends and colleagues in the Research and Development Group who helped me expand my practical knowledge.

My family and friends have been instrumental in my success. I thank my cousins, aunts, uncles, and grandparents (some now in absence), who have all been

very supportive throughout the years. I also thank my college friends and roommates who made my time at UCLA so enjoyable and provided me with some much needed distractions. The same goes to my friends from high school who have always believed in me.

Finally, I would like to thank those closest to me. My parents and sister Negin, for their unconditional love, support, and encouragement, my new brother-in-law Greg, and my Sara, whose strong support and smiles helped me get to the finish line.

VITA

1976	Born in Glasgow, Scotland
1997	B.S. (Hon) Computer Science and Engineering University of California, Los Angeles Los Angeles, CA
1998-1999	Teaching Assistant University of California, Los Angeles Los Angeles, CA
1999	M.S. Computer Science University of California, Los Angeles Los Angeles, CA
1999-2001	Software Engineer LiquidMarket Inc. / NBC Internet Los Angeles, CA

PUBLICATIONS AND PRESENTATIONS

1. N. Aghdaie and Y. Tamir, "Efficient Client-Transparent Fault Tolerance for Video Conferencing," *Proceedings of the 3rd IASTED International Conference on Communications and Computer Networks*, Marina del Rey, CA (October 2005).
2. N. Aghdaie and Y. Tamir, "Fast Transparent Failover for Reliable Web Service," *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, pp. 757-762 (November 2003).
3. N. Aghdaie and Y. Tamir, "Performance Optimizations for Transparent Fault-Tolerant Web Service," *Proceedings of 2003 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada,

pp. 29-32 (August 2003).

4. N. Aghdaie and Y. Tamir, "Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support," *Proceedings of the The 11th International Conference on Computer Communications and Networks*, Miami, Florida, pp. 63-68 (October 2002).
5. N. Aghdaie and Y. Tamir, "Client-Transparent Fault-Tolerant Web Service," *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, pp. 209-216 (April 2001).

ABSTRACT OF THE DISSERTATION

Transparent Fault-Tolerant Network Services Using Off-the-Shelf Components

by

Navid Aghdaie

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2005

Professor Yuval Tamir, Chair

The growth of the Internet has led to the development of critical network services where erroneous processing or outages are unacceptable. The availability and reliability of services such as online banking, stock trading, reservation processing, and online shopping, have become increasingly important as their popularity grows. Downtime and failures lead to unsatisfied customers and translate directly into lost revenue for the service providers.

Fault-tolerance techniques use redundant components and/or redundant processing to ensure continued correct operation despite component failures. Most existing fault-tolerance solutions for network services do not provide fault-tolerance for active connections at failure time, expect servers to be deterministic, or require changes to the clients. These limitations are unacceptable for many current and future network service applications. We propose a methodology for providing fault-tolerance without the limitations mentioned above. Our solution, based on a standby backup approach, is transparent to the clients and requires minimal changes to the server OS and application.

We have used our methodology to add fault-tolerance features to two popular

types of network services — web service and video conferencing. Off-the-shelf hardware and software components were used as the basis for both implementations. Modifications to the OS network stack using Linux kernel modules provide fault-tolerance at the connection level. At the application level, modifications to the web server and multi-conferencing unit, respectively, provide application-level synchronization and allow handling of non-deterministic server behavior. The associated issues, challenges, and tradeoffs of our methodology are presented in this work. The evaluation of our prototype implementations shows that client-transparent fault-tolerance can be achieved with relatively low overheads.

Chapter One

Introduction

The increase in accessibility of the Internet to the masses and recent advancements in technologies such as broadband and wireless communication has led to the development of a growing number of network applications and services. Online banking, stock market transactions, airline reservation processing, retail shopping, tax payments, and text messaging are just a few examples of services now available over the Web. These services are becoming increasingly popular and adapted into our culture and everyday lives. As the growth and adaptability of these services continues to accelerate, users will become increasingly dependent and expect online services to be reliable and always available. Hence, an increase in the availability and reliability of a service will translate directly into increased customer satisfaction and profits for the service provider.

This expected trend of increased demand for highly available and reliable network services is consistent with the history of other “utilities” such as telephony systems. At first, users are ecstatic to gain access to the service. Failures and downtimes are mere inconveniences to be dealt with. As the technology matures however, the users expectation of reliability grows. Hence, large efforts were placed into making the telephony systems more reliable [Keis64, Ketc65]. The same can be expected from network services. Users are already becoming intolerant of outages and errors once common at the Web’s infancy. In order to meet these increased expectations, extensive use of fault tolerance will be required. Since the competition is often so easily accessible — only a click away

— it can be argued that the motivation for building highly reliable network services is in fact greater than other “utilities.”

High availability and reliability has been the focus of a significant amount of research and has led to development of some products. However, there are several drawbacks that are prevalent in much of the existing solutions. Many solutions [Ande96, Andr96, Dias96, Cisc99a, Cisc99b, Inte00, RND 1, Aver00, Best98] focus on increasing only the availability of a service and not its reliability. Redundancy is used to provide service to new clients that arrive after a fault. However, the service state for active clients is not preserved and is lost if a fault occurs.

Existing solutions that do increase reliability and recover active connections often require changes to the clients [Snoe01, Fro100]. Because of the large number of already deployed clients without reliability features (e.g, web browsers), and the general tendency for the clients to be generic and not under the control of the service provider, it is beneficial for reliability solutions to be client-transparent. No client knowledge regarding replication or reliability features should be required.

In this work, we present a methodology for increasing the reliability of network services in a manner that is transparent to clients. We demonstrate the reliability enhancement of two prototype off-the-shelf applications based on our methodology and investigate the associated trade offs involved when building such systems.

1.1. Network Services

Network services are potentially composed of many different elements which can be organized in several alternative ways. The architecture of the service influences the way fault-tolerance and reliability features for the service are implemented. Hence, it is important to begin with a high level understanding of widely adopted architectures commonly used by network service providers.

In its simplest form a network service uses the client-server paradigm. The client is the user of the service, and the server provides the requested service. The communication between the two typically uses a request-reply protocol. A user establishes a connection to a server and sends a request. The server receives the request, processes it, generates an appropriate reply, and sends the reply back to the client.

For Internet services, client devices are diverse (e.g., PCs, mobile phones, etc.) and typically operate independently of the service. The client application is also usually developed independently of the service, and tends to be general purpose applications that can access many different services. At this time, the most common client application used to access network services is a web browser.

Servers receive client requests, process them, and generate replies. Hence, most of the processing and data storage required by the service occur on the server. The server is typically controlled by the service provider and a single server may simultaneously provide service for many clients.

Although a wide variety of communication protocols are possible, the predominant protocols used on the Internet are TCP/IP and HTTP. TCP provides reliable transmission of messages using mechanisms such as sequencing and

retransmission. HTTP is a one-to-one request-reply protocol which requires reliable transmission, hence, it is typically implemented on top of TCP.

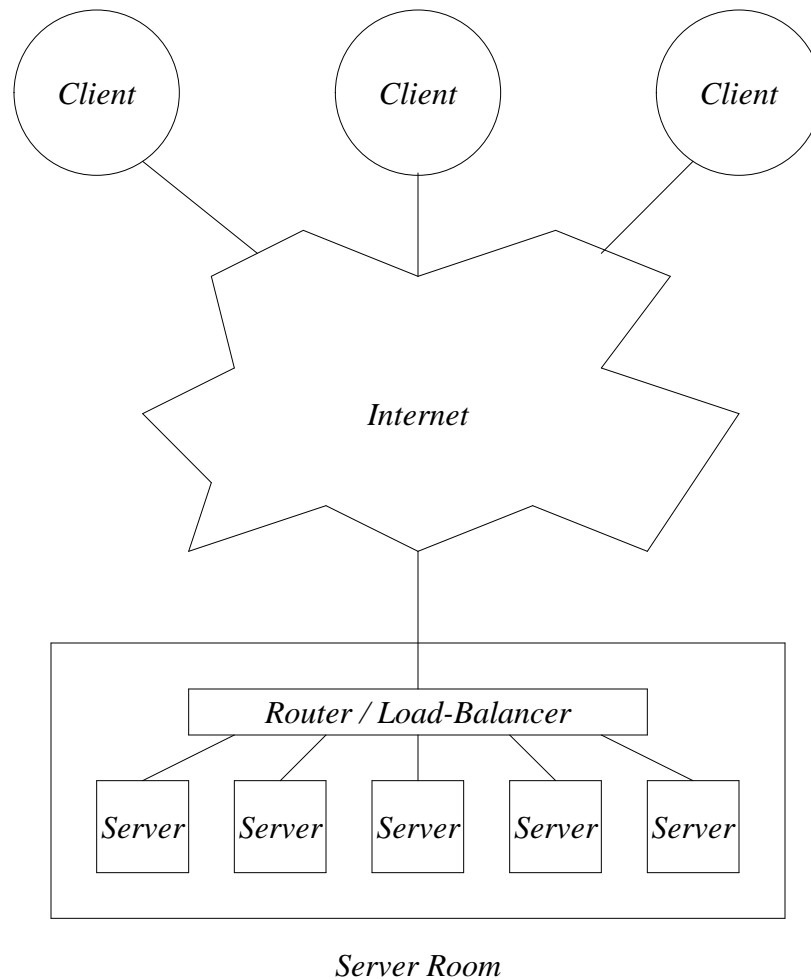


Figure 1.1: Typical Organization of Network Service Elements

For most services, a single server host does not contain sufficient resources (e.g., CPU cycles, memory, etc.) to provide service for a large number of clients. Alternate resources (i.e., server hosts) must be added to the service by the service provider in order to achieve the desired service throughput. These server-side

resources must be organized in a scalable architecture with key factors being high performance, low cost, and ease of maintainability.

Figure 1.1 shows a common architecture for network services. A pool of independent server replicas is maintained by the service provider in a single location, commonly referred to as a “server room”. Each server is a replica in the sense that it can provide the same service. Distributed storage or data replication schemes are used to ensure that each server replica can access the same identical data. Load-balancing techniques are typically used to distribute the client requests amongst server replicas. Hence, the server pool will be capable of handling a greater number of client requests, increasing the service’s throughput and availability.

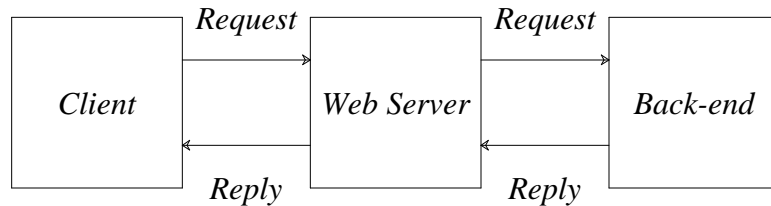


Figure 1.2: Three-tier Web Service Architecture

Network services are often implemented as multiple tiers of servers (Figure 1.2). For example, Web services are typically provided using a three-tier architecture, consisting of: clients, front-end servers, and back-end server. The division of a service into these components increases the ease of maintainability by allowing for the independent and modular development of the service sub-systems: clients for handling user interaction, front-end servers for request routing and application protocol implementation, and the back-end servers for providing stable storage and transaction processing.

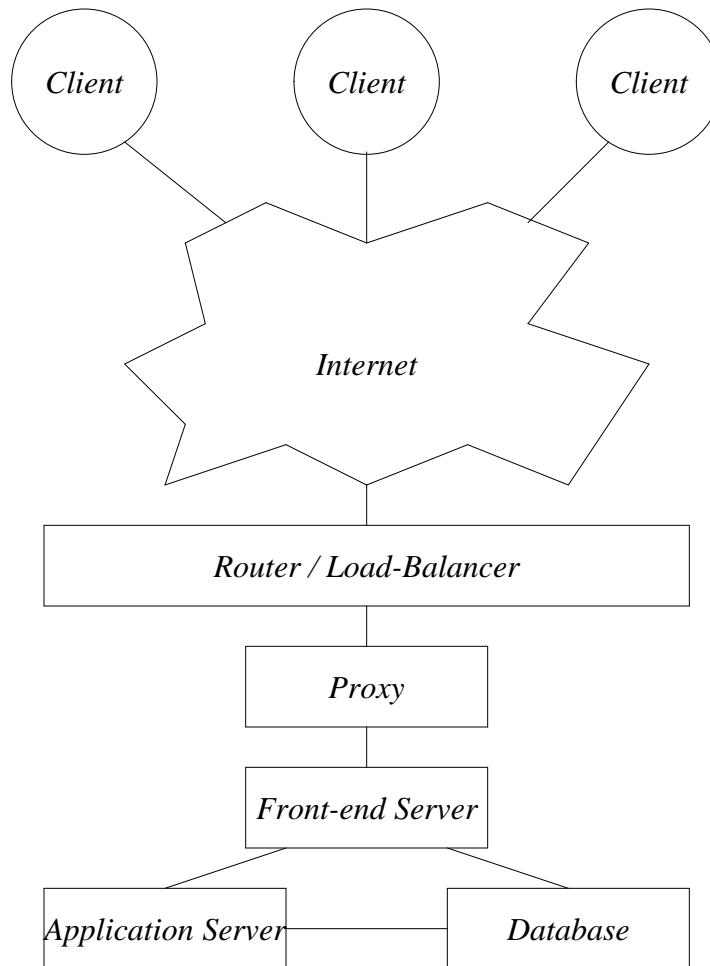


Figure 1.3: Multi-tier Service Architecture

The three-tier model can be extended into an n-tier architecture (Figure 1.3) where additional service elements compose new tiers of service. For example, proxies may be used to add new functionality such as content caching [Wess99], improved request routing, or service content adaptation for specific client devices [Schi01]. Another common example is the use of a front-end server as a workload distributor. Tasks associated with a single client request are identified by the front-end server and distributed among multiple back-end servers. The

processing of the tasks can take place in parallel, increasing the system performance and scalability. The results from the back-end servers are then passed back to the front-end server, which combines them in a presentable format and sends a single reply to the client.

1.2. Service Availability and Reliability

Service users often desire high availability, i.e., readiness for correct service, and high reliability, i.e., continuity of correct service [Aviz04]. The failure of service components can result in a service outage and in turn a reduction in the overall service availability and reliability. The causes of service failures can be divided into three types: network faults, client faults, and server faults.

1.2.1. Network Faults

Network faults encompass link and network element failures. A router may fail or a wire may be cut. There has been extensive research on network reliability and many fault-tolerance solutions are available. For example, redundant paths are typically built in the networks to tolerate link failures. The network elements are typically stateless and fault-tolerant protocols and recovery procedures have been developed. In addition, end-to-end reliability protocols such as TCP are commonly used to tolerate intermittent faults.

1.2.2. Client Faults

Client faults involve the failure of the user devices that are used to connect to the network service, or operator errors. Since these devices are typically maintained by the users and are not under the control of the service providers, we do not cover them. Building reliable client devices and applications and handling operator errors are important issues, but are beyond the scope of our research.

1.2.3. Server Faults

The focus of this work is the transparent handling and recovery from server faults. A server may fail due to the failure of one of its key components such as memory, hard-drive, or power supply. Server faults may also occur due to a “site failure” where the entire server hosting location fails, for example due to a power failure or terrorist attack. Geographical replication solutions [Shen00] are required to handle this type of failure. Our work focuses on recovering from faults that occur within a “server room.” We assume faults to be isolated to a single server host and use replication techniques with servers on the same subnet. Most of the existing techniques used for handling server faults do not provide fault tolerance for requests being processed at the time of server failure or they require deterministic servers. These limitations are unacceptable for many current and future applications of the network services.

1.3. Client-Transparent Fault-Tolerance

The main contributions of this work are a design methodology for adding client-transparent fault-tolerance to off-the-shelf implementations of network services, and the evaluation and validation of prototype implementations based on this methodology. Fault tolerance requires the ability to use alternate resources if some of the resources being used fail. This is achieved by having replicated (redundant) resources. The alternate resources must have a copy of up-to-date state so that they can continue to provide correct service. As discussed in Chapter 4, the basic approach used in this work is based on a combination of two well known techniques: *active replication* and *warm standby sparing*. With active replication (*hot standby sparing*) all processing is done by multiple replicas so that a spare (backup) replica has up to date state and can take over processing with minimal interruption. With warm standby sparing, the spare (backup) is active but does not perform all the processing of the primary replica. Instead, the state of the backup is kept nearly up to date using state updates from the primary. Neither of these approaches is new and the ideas behind them have been used in fault tolerance solutions for many decades [Keis64, Toy78]. The challenge is in applying these techniques to the emerging network services field.

The main challenges for employing replication for network services (discussed in more detail throughout this dissertation) include:

Client Transparency — Since clients may be developed independently of the service or be already widely deployed (e.g., web browser or mobile phone), it is essential to minimize any requirements from the clients. Standard clients without any knowledge regarding replication or reliability features should be able to

communicate to and use the service using standard protocols such as TCP. Furthermore, server failures should not cause any active client connections to fail. At worst, the client should experience a minimal interruption to the service.

Handling of Non-Determinism — Simple replication requires the processing of requests to be deterministic, i.e., processing of identical inputs to the system must result in the production of identical output. While this condition typically holds for most requests, in many practical systems it does not *always* hold. For example, the service application may return the time of day or may use a random number generator to determine the order of items on a list that it returns. Thus, the processing of the same request on a different host or at a later time will not always result in the generation of an identical reply. If this is possible, the service is *non-deterministic*. In order to use replication with practical applications, sources of non-determinism must be identified and the non-deterministic state changes on the replicas must be synchronized.

Reusability — Given the complexity of the implementations of many servers, it is advantageous if a solution can be used with existing off-the-shelf server implementations with minimal changes.

Efficiency — A fault tolerance solution should minimize overhead during fault-free operation and have minimal interruption time during failover and recovery from faults. For example, duplication typically incurs an overhead of at least 100%. However, identification of critical state and replication of only those key components can result in duplication overhead of less than 100% (as shown in Section 4.6 and Chapter 5). Implementation choices, such as which aspects of the scheme are implemented inside the kernel versus at user-level, can also have a

large impact on the efficiency of the replication solution.

We have employed our methodology in two specific examples of network services: a web service based on the multi-tier architecture discussed above, and a monolithic centralized video conferencing server. Web services are transaction based. Communication takes place over reliable protocols (e.g. TCP) and it is expected that every request will receive a reply. Hence, it is imperative for the fault tolerance solution to not lose any information and insure that *every* request receives a correct reply in spite of a fault.

Video conferencing is typically implemented using a centralized conferencing server. The server routes and adapts the media received from each client to other clients participating in the conference. Without fault tolerance features, the failure of the conferencing server will result in the loss of communication between all conference participants. The reliability requirements for video conferencing and in general multimedia services contain both similarities and differences from those of a web service. In both cases critical state must be preserved (via replication) to allow for seamless recovery from failures. The difference is that multimedia services typically contain a combination of reliable and unreliable communication. As discussed further in Chapter 5, reliable channels (e.g. TCP) are used for control, while unreliable streams (e.g. UDP) are used for the media. Hence, some unreliability in communication is inherently allowed and loss of some information (i.e., media) due to a fault is acceptable. The goal for the reliability solution is to minimize the loss and thus reduce the interruption time noticed by the client.

1.4. Organization of this Work

The rest of this dissertation is organized as follows. We begin by presenting existing availability and reliability solutions for network services and related research work in Chapter 2. We discuss solutions aimed purely at increasing availability, client-aware solutions, which unlike our work, have the drawback of requiring changes to clients, and advancements in client-transparent server fault-tolerance which have occurred concurrent to our work.

Chapter 3 presents our design methodology for adding client-transparent fault-tolerance to off-the-shelf implementations of network services. We identify several key characteristics of network services that should be taken into account in designing fault tolerance schemes for these services. We show how to develop a fault tolerance scheme as a composition of mechanisms that target or take advantage of these characteristics, and discuss the associated issues and tradeoffs.

We have demonstrated our approach through proof-of-concept implementations and their evaluation. Chapter 4 presents CoRAL, a client-transparent fault-tolerant web service developed based on our design methodology. CoRAL (**C**onnection **R**eplication, **A**pplication-level **L**ogging) actively replicates the communication connection state and uses application-level logging to preserve user-level messages. CoRAL does not require any changes to the clients and can handle non-deterministically generated server replies. We have measured the performance overhead during fault-free operation (in terms of latency, throughput, and processing cycles) and the recovery time from failures. In addition, fault injection experiments were used to validate our fault tolerance scheme. The design, implementation based on Linux and Apache, and performance

measurements and optimizations are discussed.

Chapter 5 discusses the use of our methodology for increasing the reliability of a video conferencing server. Fault tolerance features are added to an off-the-shelf implementation of video conferencing based on Linux and a H.323 multi-conferencing unit application. Our implementation combines kernel modules with small changes to the server application to efficiently preserve both the reliable connections used for control messages and unreliable connections used for media transfer. While the scheme is based on replication, the associated overhead is negligible since the backup does not process the media streams. We present the performance overhead during fault-free operation and impact of server failures on the service provided to the clients.

Chapter Two

Related Work

This chapter presents previous research work and existing products related to the increasing of availability and reliability of network services. Early work in this area focused on increasing only the availability of services. Replications protocols for stateless elements [Li98, Knig98] (e.g., routers) and load balancers [Aver00, Cisc99b, Inte00] are examples.

Recent research in the area has focused on increased reliability and correct handling of every client connection and request, as is the goal with our work. Some solutions are client-aware and require changes to the client OS [Snoe01] or application [Zand02]. Other research, conducted concurrently with our work, include similar high-level requirements as our work and use client-transparent approaches to increase service availability.

In the rest of this chapter we look at the related work, discuss the differences in approaches, and present the advantages and drawbacks of each solution. Section 2.1 presents standard techniques used for increasing the availability of network services. Section 2.2 presents solutions that increase *both* the availability and reliability of network services. It includes discussion of solutions that are client-aware (requiring changes to the clients) and client-transparent fault tolerance solutions which share our goals. Section 2.3 discusses solutions that increase the reliability of only some elements (i.e. backend servers) of a service. Finally, Section 2.4 concludes the Chapter by looking at fault-tolerance solutions designed specifically for video conferencing applications.

2.1. Availability Solutions

In this section we describe solutions which increase the availability of network services. The goal of this class of solutions is to ensure that in the event of a server failure, future client requests are routed to and serviced by an alternate server. These solutions do not have a requirement to correctly process every client request. Hence, some client requests may be lost or not processed if a server fails. Presented in the rest of this section are several techniques that can be used to increase service availability. These techniques differ in the mechanisms used to route requests to different servers.

2.1.1. DNS Based Load Balancing

The DNS mechanism used to map host names to IP addresses can be used to route new requests to working servers. A client is typically aware of the host name where its request is to be sent. This host name must first be resolved to an IP address via DNS, before the client can actually send the TCP/IP packets containing the request. It is possible for the DNS system to increase the availability of the system by detecting faulty servers and not returning their addresses in response to DNS queries.

If the service is composed of a pool of server replicas, Round Robin DNS [Bris95, Katz94] can be used to increase availability by changing the host name to IP address mapping depending on the state of the system. In standard operation, Round Robin DNS replies with the IP address of a different replica for each request, in a round robin fashion. As a result, client requests are load balanced amongst the replicas. When a server failure is detected, the advertised

host name is no longer mapped to the IP address of the failed server host. As a result, new client DNS requests arriving after a failure will not be given the IP address of failed servers.

The DNS aliasing method [Bris95] is a similar alternative to Round Robin DNS. Instead of the DNS system returning a different IP address each time, it returns several IP addresses with each DNS query. All of the addresses correspond to identical server replicas, and the client can choose to send its request to any of the addresses. Similar to Round Robin DNS, increased availability can be achieved by removing the IP address of faulty server hosts from the list of aliases. Alternatively, the client can choose to retry its request to another one of the IP addresses should its first try fail.

DNS schemes require the client to re-issue its service request if it does not receive a reply for a request. In practice, clients may continue to see the old mapping due to DNS caching at the clients and DNS servers. This reduces the effectiveness of DNS schemes since it may take a long time for DNS caches to be updated.

2.1.2. Centralized Routers and Directors

It is possible to get new requests to working servers by sending them first to *routers* (or *directors*) that are aware of the state of the servers and forward requests to working servers. Conceptually, centralized routers [Ande96, Cisc99a, Cisc99b, Inte00] are similar to DNS based schemes. However, centralized routers are more effective at increasing availability since every packet passes through the central router and is subjected to the scheme. As mentioned previously, DNS schemes

have the drawback of having stale addresses (of failed servers) due to DNS caching at the client or upstream DNS servers. This problem does not exist with centralized routers since any decisions regarding failure detection or corrective measures will be visible starting with the next arriving packet.

Centralized router implementations are typically based on TCP/IP packet header rewriting. The service address is mapped to the router and clients send their requests there. Hence, incoming packets from clients contain the router's address as the destination. Once the packets reach the router, the destination address is changed to the address of the server that has been selected by the router. Also, the packet source address is changed to be the router's address, and the packet is forwarded to the selected server. As a result, the server sees a packet as if the router was the client. Therefore, the server uses the router's address as the destination in reply packets that it generates, and the reply packets are sent to the router. The router maintains mapping tables of client and server addresses which keeps track of corresponding client-router and router-server connections. Once server reply packets reach the router, the mapping table is consulted and the destination address is to the appropriate client's address. The source address of the packet is also changed to be the router's address, so that to the client it appears as if the router is the server and is sending it a reply. The router also checks the mapping table for incoming client packets, and routes future packets from the same client connection to the same server.

Centralized router and director schemes are typically used for load balancing purposes. Hence, in addition to forwarding requests only to servers that are operational, the router's decision regarding server selection also typically includes

a load monitoring component. The loads of a pool of replica servers are monitored by the router, and the client request is forwarded to the least loaded server. Other selection algorithms such as those considering request locality (routing of identical requests to the same server) are also used.

The central router is a single point of failure and a performance bottleneck since all packets must travel through it. Distributed Packet Rewriting [Aver00, Best98] is a distributed implementation of some of the functionality that is achieved by centralized routers. It avoids having single entry point by allowing each server to act as a router, and having servers send their outgoing messages directly to clients. Some of the router logic is implemented in each of the replica servers. Each client is aware of the address of only one of the replica servers in the pool. The distribution of addresses to clients requires running a load-balancing scheme such as Round Robin DNS on top. When a server receives a client request, it decides whether to process it locally or route it to an alternate replica. As with the centralized schemes, the decision can be based on the load as well as on which of the server replicas are operational. The source address in outgoing server packets will always contain the destination address used by the clients. Hence, from the view point of the client requests are processed by a single server although in actuality they may have been routed to an alternate.

2.1.3. Application-Level Redirect

With appropriate support in the application-level protocol, it is possible to cause the client to *redirect* requests to working servers. For example, the HTTP protocol [Bern96] contains a redirect directive that enables a server to direct the client to try its request at an alternate location. A load balancing scheme can take advantage of this feature by implementing a “redirector” node [Sury00]. Clients initially are only aware of the redirector’s address. Hence, all client requests are sent to the redirector. Upon the receipt of each request, the redirector chooses a server and redirects the client to retry its request to that server. In some implementations, the redirector can also act as a server and choose to process some of the requests itself.

Similar to centralized routers, the application-level redirection approach can be extended to provide enhanced availability by adding fault detection to the server selection component of the redirector. The difference is that instead of directly forwarding the client requests to the selected server, a redirect directive is sent back to the client directing it to try its request at the selected server. To increase availability, the redirector monitors the servers and does not redirect clients to any faulty servers. Hence, faulty servers are effectively removed from service and the overall availability of the system is increased. The redirector itself is a single point of failure in this scheme. Therefore, there is an assumption here that the redirector does not fail, or that it fails less frequently than the servers due to its relatively simpler functionality.

2.2. Availability and Reliability Solutions

Solutions that increase reliability as well as availability have stricter requirements than availability only solutions. The main goal of availability only solutions is to ensure that the service is quickly restored after a failure so that new incoming clients can be serviced. With these solutions, a server failure will often cause some of the client requests to be lost. As a result, some clients may not receive a reply for their request or they may receive incorrect replies.

Solutions that provide increased reliability extend the requirements and ensure that *every* client request is processed correctly, and a correct reply is sent to the client. Replication is used to preserve the relevant server state over failures, allowing for the recovery of requests whose processing was “in-progress” at failure time.

Implementations of availability and reliability solutions can be categorized into two groups: client-aware and client-transparent. Client-aware solutions require changes to the clients at either the OS level [Snoe01, Sult01] or application level [Frol00, Zhao01]. Clients are active participants in the fault-tolerance scheme and their interaction with servers may differ from the communication performed with standard off-the-shelf servers. Client-transparent solutions do not require any additional or different operations from clients in support of the reliability enhancement scheme. Thus, no changes to the clients are required.

The redundant resources of a service can be co-located or be geographically distributed. Geographically distributed solutions are resilient to failures caused by complete destruction of a server site, such as a terrorist attack or natural disaster. However, the performance overhead is typically larger because communication

between replicas must take place over the Internet. Also, on the Internet, IP addresses are typically mapped to a static geographical location. Client packets sent to an IP address will be routed to the same geographic location unless entities which control the Internet backbone routers make changes to the routing tables. Since service providers are unlikely to have access to Internet routers, client-transparent solutions cannot cause client packets sent to the same IP address to be routed to a different location after a server failure. Hence, client-transparent schemes require either co-location [Aghd01, Aghd02] or a redirector [Shen00] that is a single point of failure to the system.

We discuss the details of schemes which increase both availability and reliability in the rest of this section.

2.2.1. Client-Aware Solutions

Several client-aware schemes [Frol00, Zhao01, Snoe01, Sult01, Zand02, IBM 2, Evan03, Bilo03] have been proposed for increasing the availability and reliability of network services. They typically involve the client in a fault recovery protocol. The client is aware of server-side replication and is actively involved in the fault-tolerance aspects of the system. In the rest of this section we discuss these schemes in detail.

Frolund and Guerraoui [Frol00] increase reliability and recover in-progress requests by using replication and a distributed protocol. Here, a three-tier server structure is considered, and a protocol is developed to assure the exactly-once execution of a client transaction. The developed protocol is similar to the two-phase commit protocol with clients, as well as application servers (web servers)

and backend servers, being active participants. Replicated application servers receive a request from the client, and first send a “prepare” message to the backend. The applications servers then reach consensus on the reply from the backend, and move to the commit phase of the protocol. Subsequently, a commit or abort message is sent to the backend and the client. The application servers use “write-once registers” for synchronization and reaching consensus. The drawback of this scheme is that the client must retransmit the request to multiple servers upon failure detection and must be aware of the address of all instances of replicated servers. A consensus agreement protocol is also required for the implementation of their “write-once registers” which could be costly, although it allows recovery from non fail-stop failures.

Zhao et al [Zhao01]. have described a similar infrastructure for increasing reliability in a three-tier systems. Their implementation is based on CORBA. The front-end application server is actively replicated. A two-phase commit protocol is used for the transaction processing communication between the front-end application server and the back-end database. A user-level library is added to the clients, enabling automatic failover to a replica application server in the event of a failure.

Snoeren et al [Snoe01] developed a scheme that migrates a faulty server’s active client connections to a working replica. The scheme is based on the replication of critical server state maintained for each connection (e.g., request URL and TCP sequence number) using periodic dissemination of the state in a weakly consistent manner, and the ability to migrate active connection end points. Server replicas are organized into “support groups.” A health monitoring agent,

implemented at either the client or server site, detects failures and notifies each server in a support group when a fault occurs. A server in the group is selected as the destination host, and all active connections are migrated to that server. The implementation is transparent to the application, however, kernel-level modifications at all clients and servers are required for the implementation of connection migration capabilities. Changes to the TCP state machine are proposed [Snoe00], allowing a host to seamlessly migrate the transport layer state of an active connection to a different host. The migration can be initiated by the local host (e.g. server) or remotely (e.g. a client). In addition to the kernel-level changes, a “wedge” or proxy on each server host relays TCP connections from the clients to the local server. The soft-state maintained by each wedge is periodically examined and information about each connection is sent to the the server support group (i.e., replicas). Thus, if a server fails, a replica in its support group will have both the transport layer state and applicable application level state, and will be able to seamlessly take over the active connections.

Migratory TCP (M-TCP [Sult01]) is another client-aware scheme similar to Snoeren [Snoe01]. The client can migrate the remote end-point of an active connection from a faulty server to an alternate replica. Changes to the transport layer protocol are used for the implementation. M-TCP is designed for increasing system performance and lacks the synchronization of server-state that is required for failover. Both server replicas are involved in the migration of applicable server state and thus must be working correctly for the migration to take place. Hence, currently M-TCP only increases service availability. However, it can be transformed into a reliability solution with the addition of a server state

synchronization on top. In general, the requirement to use a specialized transport layer protocol at the client is a major drawback. It is obviously difficult to deploy these solutions on today's standard clients (e.g., web browser on a PC). Therefore it is unlikely for these solutions to be widely used, unless the required client kernel modifications become part of a standard and widely adopted into off-the-shelf operating systems.

Reliable network connections [Zand02] is another client-aware connection based migration based solution. The difference with this solution is that it only requires user-level modifications. The implementation uses wrappers around socket calls on both the client and servers. The client is capable of detecting server faults. If a failure is detected, the implementation migrates the connection to a replica and retries the requests there.

HTTPR [IBM 2] is a protocol built on top of HTTP [Bern96] for the reliable transport of messages from one application to another over the Internet, despite the occurrence of failures on either end. HTTPR messages are encapsulated within the payload of HTTP request and reply messages. Globally unique identifiers, composed of client URI (Uniform Resource Identifier), server URI, and a channel identifier, are created for each connection. HTTPR also defines a set of commands that allows each side to send a transaction request or query the other side regarding the status of previous transactions. If a server fails, the client can issue a "report" command to the (recovered) server and based on the reply from the server determine exactly which messages have been safely received by the server. Since HTTP requires in-order message processing, the client can safely retry the lost messages. Also, the server can discard messages that the client reports as having

been safely delivered. This scheme requires both the client and server application to implement the HTTPR protocol.

Web Services Reliability Specifications [Evan03] describes a SOAP based protocol for exchanging SOAP messages with guaranteed delivery without duplicates. Although SOAP messages are independent of underlying protocol, they are typically sent on top of HTTP. This scheme requires the client to log all outgoing messages, and the server to log all incoming messages. Every reliable message contains a globally unique ID. Upon the receipt of a request, the server must send a reply message (i.e. an HTTP message) back to the client. The reply message is either an acknowledgement or a fault message. If a client receives a fault message or if it does not receive an acknowledgement for a request within a timeout period, it must resubmit the request using the same ID. The globally unique ID is used to eliminate duplicate message, and sequence numbers are added to messages by the clients to assure guaranteed message ordering.

In general, although client-aware solutions provide flexibility for fault-tolerance solutions, unless client-aware standards such as recent industry proposals WS-Reliability [Evan03], WS-ReliableMessaging [Bilo03], or HTTPR [IBM 2] are widely accepted, client-transparent solutions are imperative.

2.2.2. Client-Transparent Server Fault Tolerance

In order to hide server failures from the client, the system must replicate the critical server-side state, detect failures, recover from them, and continue operation and communication using a consistent (same as pre-fault) address — all without any explicit help from the client. Over the last few years, several schemes that

achieve client-transparent fault tolerance have been proposed [Shen00, Alvi01, Zago03, Burt02, Luo01, Marw03, Koch03]. The common features of these schemes are client transparent replication of server side state and recovery and failover of active communication connections from a failed server to a working replica. The key differences among the schemes are implementation choices, efficiency of the scheme such as failure-free overhead and recovery time, and assumption, e.g., deterministic versus non-deterministic server behavior. In the rest of this section we discuss these schemes in detail.

2.2.2.1. HydraNet-FT

HydraNet-FT [Shen00] is a scheme that uses active replication to preserve server state. Two (or more) server replicas process every TCP packet, locally producing an up-to-date copy of the server state required to maintain communication with the client. Client packets travel through a specialized router (“redirector”) which sends a copy of the packet to each of the server replicas. Each replica independently processes the packets and generates acknowledgements. The acknowledgements generated by the backup(s) are routed to the primary, and the primary sends the actual acknowledgement to the client only after it has received a copy from the backup. This assures that the client packets are not acknowledged unless all replicas have received a copy. Hence, if a packet is lost enroute to one of the replicas, the client will not receive an acknowledgement and will retransmit the packet.

In normal operation, only the primary sends generated reply messages to client. Replies generated by other replicas are effectively dropped. Client

acknowledgement packets are broadcast to the replicas by the redirector in the same manner as client data packets. Since all replicas see all client packets and generate the same replies, a backup can takeover at any time should the primary fail.

HydraNet-FT incurs a large processing overhead because all of the processing is duplicated on each replica. In addition, due to the use of simple active replication, the servers are required to be deterministic. An advantage of this scheme is that unlike most other client transparent schemes, the server replicas can be geographically distributed anywhere on the Internet. The use of redirectors that provide multicast functionality allows for this flexibility. However, the use of redirectors is also a weakness of this scheme since they are a single point of failure and a bottleneck, similar to the centralized router schemes.

2.2.2.2. FT-TCP

Alvisi et al. implemented FT-TCP [Alvi01], a kernel level TCP wrapper that transparently masks server failures from clients. FT-TCP uses the log and replay approach. Incoming client packets are logged (redundantly stored) on an alternate host. If the server fails, logged packets are replayed (reprocessed) to regenerate the server state prior to the fault. Once the server state is restored, the communication and processing of requests can resume.

FT-TCP is implemented using layers of software around the server TCP stack and a logger on a separate processor is used to log the packets. Incoming client packets arrive at the server and are logged to the logger. The logger sends an acknowledgement to the server when it receives and logs the packets. The

incoming client packets are acknowledged (by the server to the client) only after they have been logged. As mentioned above, if the server fails, all logged packets are replayed from the logger to a new server. As a result, the server state is recovered to the same state as prior to the failure and communication can continue.

The basic log and replay approach requires deterministic servers in order to ensure that the state obtained from the reprocessing of the packets will be identical to the state prior to the fault. The FT-TCP implementation includes some support for non-determinism. For example, since the initial sequence number selection in TCP is not deterministic, a recovered server may produce different initial sequence numbers than the failed server. The FT-TCP wrapping layers handle this non-determinism by also logging the initial sequence number and performing operations on packet TCP sequence numbers to assure transparent failover and recovery. FT-TCP also logs the number of bytes returned by the `recv` system call, thus handling any non-determinism that might cause changes to the behavior of that system call.

Since all packets received during the lifetime of each connection must be replayed in case of a failure, FT-TCP can suffer from long recovery times. A follow-up version of the FT-TCP scheme [Zago03] contains a few changes and improvements: A backup host is used instead of the logger, more sources of non-determinism are handled via the synchronization of system calls between the primary and backup hosts, and the system supports operation in either cold (logging) or hot (active replication) modes. The hot replication mode has shorter recovery times at a cost of higher processing overhead during failure-free operation.

2.2.2.3. HotSwap

HotSwap [Burt02] is an active replication scheme that uses a similar approach to FT-TCP [Zago03]. TCP connections are replicated on a primary and backup server. Both servers see all packets that are sent by clients. Both servers execute identical copies of the application. In normal operation, the outgoing packets from only one of the replicas (primary) is sent to the client. HotSwap handles non-deterministic behavior by ensuring that applications on both servers are synchronized at each system call. Identical system call results lead to identical (deterministic) application execution and thus identical state on both server replicas.

HotSwap suffers from the inherent costs of active replication as both primary and backup must perform duplicate application processing. The synchronization of system calls adds additional overhead, which can be significant specially in terms of latency.

2.2.2.4. Zero-Loss Web Services

Luo and Yang's work "Constructing Zero-Loss Web Services" [Luo01] uses an alternate approach for implementing the multicast (from client to server replicas) that is required for replication. It also handles non-determinism via a store and forward approach. IP address aliasing is used to multicast the client requests to both the primary and backup servers. Instead of forwarding the packets from one host to its replica [Aghd01, Aghd02, Shen00, Zago03] Both servers read the packets whose destination address is the service's address. Although this avoids the typical multicast associated overheads, such as extra messages, relying

on IP aliasing for the multicast of client packets is not completely reliable. This approach does not guarantee that both primary and backup servers will have a copy of a client packet before an acknowledgment is sent to the client. Hence, there is a possibility that only one of the servers would receive a packet since packets may be dropped due to buffer overflows in one of the servers, or due to physical network conditions. Therefore, it leaves a possibility for the two servers to have different states and thus, some failures may not be handled. Another disadvantage of IP aliasing is that it requires the use of a shared network.

For content that is dynamic and generated non-deterministically, this work relies on a distributor and a “store and forward” approach. The distributor receives a client request and sends it to a server. The server reply is sent back to the distributor, however the distributor does not immediately forward the reply packets to the client. Instead, it stores the packets until the entire reply is received. Should the server fail in the middle of sending a reply, the distributor resubmits the request (or a partial request for the portion of the reply that wasn’t received) to a different server. As in centralized routers [Ande96, Cisc99b] and HydraNet-FT [Shen00] the distributor is a single point of failure and a possible performance bottleneck.

2.2.2.5. ST-TCP and Transparent TCP Connection Failover

Some schemes use packet snooping to implement the multicast of client packets. ST-TCP [Marw03] uses a backup which snoops client packets that are intended for a primary server. Similar to IP aliasing, snooping does not guarantee that both servers will receive all the packets. Packets may be lost on their way to

the backup or inside the backup's kernel due to overload. In order to guarantee identical receipt of packets by both replicas, the primary does not discard a packet until it receives a notification from the backup that it has received the same packet. Furthermore, to handle primary failures along with a dropped packet by the backup, a logger is required to log all incoming packets. Packet snooping will only work if the primary and backup servers are on a shared network. The snooping approach leads to lower latency overheads because packets do not have to be forwarded. However, the extra required resources (logger) make this approach a poor choice for scenarios where price/performance is important.

Transparent TCP Connection Failover [Koch03] uses a similar scheme where a backup server runs its network interface in promiscuous mode to receive client packets that are sent to the primary. The need for a logger is eliminated by forwarding backup's outgoing (acknowledgement) packets to the primary, and delaying the primary's outgoing packets until a matching packet is received from the backup. Hence, the need for additional hardware (logger) is eliminated at a cost of increasing the latency during fault-free operation.

2.3. Replicated Back-end Servers

Many database vendors such as Oracle [Orac99a, Orac99b] offer fault tolerant databases and back-end servers that provide high availability and reliability. These offerings increase the availability and reliability of the back-end server only and do not handle failures in other parts of the system. Hence, while such schemes are necessary components of three-tier architectures, they do not solve the problems dealing with front-end server failures — the main focus of our work.

The use of a fault-tolerant back-end servers in a three-tier system simplifies the implementation of a front-end solution. For example, once a back-end transaction is completed, the front-end server is not responsible for the preservation of relevant state changes that occur at the back-end.

The front-end server does have to ensure that each transaction is performed exactly once. Some requests maybe non-idempotent [Gray92], where the results of multiple executions of the same transaction is different than a single execution. To avoid repeated execution of the same transaction (e.g., retry of a transaction by a front-end server recovering from a failure), a fault-tolerant front-end server may either use a two-phase commit protocol [Fro100] or use a unique transaction ID with each transaction so that the fault-tolerant back-end can identify repeated transactions.

In summary, fault-tolerant back-end and fault-tolerant front-end servers are complimentary. A system composed of a combination of the two provides a complete reliability solution for three-tier systems capable of handling faults that may occur anywhere in the server room.

2.4. Fault-Tolerant Video Conferencing

In Chapter 5 we present our client-transparent fault tolerance scheme for video conferencing. To our knowledge, there is no published work specific to fault tolerance for video conferencing. Published work on fault tolerance features for media servers [Chu00, Chu01, Zago03] are typically concerned with servers which broadcast media to clients in a unidirectional fashion. Our work is different in that we focus on video *conferencing* — bi-directional communication among a group of

multiple clients. Video conferencing implementations typically use a stateful centralized server whose functionality is different than those of broadcast multimedia servers. However, some of the fault tolerance issues involved are related, and thus we present the details of a few media server reliability schemes in the rest of this section.

Zagorodnov et al [Zago03] adapted FT-TCP [Alvi01] to increase the reliability of Apple Darwin server which is typically used to broadcast multimedia streams to clients. FT-TCP uses the log and replay approach to mask server failures and transparently recover from server faults (see Section 2.2.2). FT-TCP includes features to handle some non-determinism (e.g., selection of initial TCP sequence numbers), but for the most part it requires the server application to be deterministic. Their experience adapting FT-TCP to work with the media server showed that the server's behavior contained additional sources of non-determinism at the application level (e.g., randomly generated session ID) which had to be addressed. To handle the non-determinism, relevant system calls were synchronized between the primary and backup to ensure the deterministic execution of the application. As a result, identical decisions are made at both server replicas and thus an identical server state is available at a backup if the primary server fails. Their results show transparent recovery from server failures and relatively low failure-free overhead due to the execution of relatively few system calls by the media server. Another scheme, Hot-Swap [Burt02], also uses system call synchronization to handle non-determinism, although that scheme has not been used for media or conferencing applications.

End system multicast [Chu00, Chu01] increases availability by using

multicast at application level. It provides overlay spanning trees for data (media) delivery. Changes to clients or deployment of proxies at strategic locations near the clients are required. In their peer-to-peer approach, each client is a node in the overlay multicast tree. Idle network resources of the clients are used to help distribute the broadcast media to other clients. With this approach, a fault at any multicast node may cause interruption to downstream clients. Faults are handled by reconstruction of the spanning tree without the faulty node. The clients are actively involved in the decision making regarding changes to the overlay network and the multicast in general. Hence, this is a client-aware solution.

Since broadcast schemes can be implemented on top of stateless routers or softstate nodes (that multicast the video), fault-tolerance solutions are relatively simpler than schemes for stateful servers. Again, the schemes presented above are based on one way delivery of video from one server to multiple clients. Our solution (presented in Chapter 5) is based on interactive communication between multiple client, using a stateful centralized conferencing server and transparent recovery from fault that may occur at the server.

2.5. Summary

In this chapter we presented research and products related to our work. Availability only solutions are the simplest and have the least overhead. They effectively remove faulty servers from the system. However, they do *not* recover active connections or in-progress requests. Thus, active clients will notice server failures, and the reliability of the service is not increased.

Solutions that increase both availability and reliability can be classified into

two categories: client-aware and client-transparent. Client-aware schemes have more flexibility in the implementation. However, since clients may already be widely deployed or independently developed, deployment of these solutions for existing applications and services will be difficult.

Client-transparent solutions share the same philosophy as our work. Server failures are masked from clients and active connections and in-progress requests are preserved over faults. Most of the existing solutions have some drawbacks, including: assumption of deterministic server behavior [Shen00], high cost [Marw03] (e.g., extra hardware for a logger), large failure-free overhead [Shen00] (e.g., due to pure active replication), and long recovery time [Alvi01] (e.g., due to log and replay). We will address and propose solutions for these problems in our work.

Chapter Three

Constructing Client-Transparent Reliable Network Services

This chapter presents a methodology for constructing client-transparent fault-tolerant network services with off-the-shelf components. We begin by presenting the requirements, targeted system model, and our assumptions. The main requirements are identification of critical service state, replication and preservation of that state over failures, error detection, and service failover. We identify the key server-side service state elements — service identity, connection state, application state — that must be preserved over server failures. We describe several approaches for meeting the rest of the requirements, explain the tradeoffs involved, and discuss the service properties (e.g. non-determinism) that must be carefully considered when adding fault tolerance features to off-the-shelf network services. A summary of our proposed methodology concludes the chapter.

3.1. Requirements for Adding Client-Transparent Fault Tolerance to Network Services

In order to provide client-transparent fault tolerance, clients must be able to continuously communicate with the service in spite of a server fault. Fault tolerance schemes for network services should increase the *availability* of the service by providing the ability to service *new* client requests following server failure. Such schemes should also increase the *reliability* of the service by ensuring that in-progress requests, i.e., those that are being processed at the time of

server failure, will be handled correctly.

In order to connect to a service, a client must have some way to specify the identity of the desired service. For example, this may be a (host name, service identifier) pair or an (IP address, port number) pair. In order to achieve client-transparent fault tolerance, the client must be able to continue to use the same mechanism and the same names to specify the identity of the service despite server failure. This implies that the implementation of the service must maintain the *service identity* despite server failure. Thus, if replication is used and the primary server replica fails, client requests transmitted in exactly the same way as before the failure must now reach a backup server replica.

Servers typically maintain some state while providing a service. For example, the server must maintain some state in order to implement a reliable communication protocol such as TCP. In order to achieve client-transparent fault tolerance, the relevant server-side state must be preserved and available on a backup server replica if the primary server fails. We classify the server state into two key components: *connection state* and *application state*. Connection state is the state required to sustain communication with clients. Application state is the state required to deliver proper service results. These state components are described in more detail in Section 3.3.

Once an error occurs, it must be detected so that recovery actions can be initiated. Error detection is typically implemented via exchanges of messages among server replicas or between server replicas and *comparators* or *voters*. With replication, if a server failure is detected, failover must occur — the system must be reconfigured to resume normal operation without the faulty server. Failover

may involve the taking over of the identity of the failed server by a replica, and the recovery and use of its preserved state.

3.2. Fault Model and Assumptions

In the worst case, faults can cause system behavior to be arbitrarily incorrect and even malicious. Such faults are called *Byzantine faults* [Lamp82]. An example of a Byzantine server fault is the transmission of unwanted or incorrect messages from the faulty server. Unfortunately, Byzantine faults cannot be handled transparently for some communication protocols. With TCP protocol for example, a faulty server may send TCP reset or FIN packets for the connection, causing the remote end (i.e., client) to abandon or close the connection. Once the client-side connection state is lost, the TCP connection cannot be transparently restored. Hence, client-aware solutions are required in order to handle Byzantine faults.

While Byzantine faults do occur in real systems, most faults that cause errors result in incorrect behavior that is not malicious. In the best case, a fault that has an impact (i.e., causes an error), causes a subsystem to die silently — stop generating outputs. In such cases, the fault is said to have caused a *fail stop* (or *crash*) failure [Schn84]. Fail-stop faults are easier to handle because the behavior of the faulty component after the fault is known — it will no longer function or generate any output. Hence, such faults can be detected using relatively simple techniques. For example, server replicas can exchange periodic “heartbeat” messages and missing heartbeats indicate that an error has occurred.

Fault injection research has shown that in practice, most faults either have no effect or they cause fail-stop failures [Made02]. Our experience with fault

injection (see Chapters 4, 5) also confirms that most failures are fail-stop. We found the most common type of failures (by far) to be process crashes. Thus, we assume the server processes to be fail-stop.

Fault tolerance solutions often assume nodes (i.e. server hosts) to be fail-stop [Aghd02, Alvi01, Shen00]. If any components of the node fails, (e.g., network interface, kernel, application processes) it is assumed that the entire node will crash. In practice, faults in a particular component of a server often do not have an effect on other parts of the system. For example, a crashed process typically does not cause a crash of the kernel or other processes. Hence, instead of assuming that nodes are fail-stop, our work is based on the assumption that *processes* are fail-stop. We assume that if a process fails, it will crash and die silently, but other processes on the host may continue to operate correctly. We further assume that a fault in the kernel that has an effect results in a host crash (fail-stop failure of the entire host).

Our scheme is based on several other assumptions besides fail-stop processes. We assume that a fault may impact only a single host at a time (a single process crash or the crash failure of the entire node). In practice faults are relatively rare. Thus the probability of simultaneous failures in independent service elements (i.e., server replicas) is order(s) of magnitude less than that of a single fault. We also assume that the server replicas are connected on the same IP subnet and that the local area network connecting the server replicas as well as the network connection between the clients and the server LAN will not suffer any *permanent* faults. In practice, the reliability of the network connection to that subnet can be enhanced using multiple routers running protocols such as the Virtual Router Redundancy

Protocol [Knig98]. This can prevent the local LAN router from being a critical single point of failure.

3.3. Service State

In order to achieve increased availability and reliability, and client-transparent seamless recovery of active connections, the server-side service state must be preserved over failures. The server-side state is composed of three main components: service identity, connection state, and application state. The details of these service state components are discussed in the rest of this section.

3.3.1. Service Identity

Service identity is the address known to and used by clients to access the service. For services built on top of the IP protocol, the service identity is the destination IP address used by the clients. The service identity also includes the TCP or UDP destination port number if those protocols are used on top of IP. If the server host that is configured with the service identity fails, a backup replica must take over the identity.

The simplest fault tolerance solutions provide increased availability for stateless service. In this case, there is no state to be preserved or repaired and there is no attempt to handle requests that are in progress at the time of server failure. Hence, only the service identity is required to be preserved. A mechanism for detecting server failure, such as heartbeats, is needed. When a fault is detected, failover is accomplished by mapping (routing) new requests to alternate resources (servers). New servers are thus allowed to take over the *identity* of the failed

server.

As discussed in Section 2.1.1, for services built on top of IP, identity take over is typically accomplished by using DNS to change the binding of server name to IP address at the client [Bris95] or by routing the packets with a fixed destination IP address to different servers at the server site. One common approach for a server site solution is the use of a special central router or load balancer that is an integral part of the fault tolerance solution [Ande96, Best98, Cisc99a, Cisc99b, Inte00]. The router detects server failures and routes client packets only to working servers. This approach does not require any changes to the server host for identity preservation. However, the router is a potential performance bottleneck and single point of failure. An alternative approach is the reconfiguration of the local network after a fault. For example, a backup replica is reconfigured with the failed server's IP address and the local network's router is informed of the location change of the IP address via ARP messages [Aghd03b].

3.3.2. Connection State

Server hosts typically maintain some state for each logical communication channel. This state is required to sustain communication with the clients. For example, TCP requires some state to be kept at each end-point in order to implement the functionality that is provided by the protocol (e.g. reliable delivery or congestion control). This state also includes partially received messages (requests) and partially transmitted messages (replies). This state is updated with arrival or departure of each packet. We refer to any communication state kept at an end-point as “connection state.”

Servers typically maintain some state for both reliable and unreliable communication. For communication protocols that do not guarantee reliable packet delivery, such as UDP, the connection state typically consists of socket structures and protocol related addressing information such as port numbers. With such protocols, packet loss is acceptable so it is not necessary to consider client packets themselves to be part of the connection state that must be preserved. It may still be useful for fault tolerance solutions to try to minimize packet loss due to faults in order to minimize the impact of faults on service performance/quality.

For a reliable communication protocol such as TCP, the connection state typically includes additional information such as sequence and acknowledgment numbers. More importantly, the connection state for reliable communication also includes any client packets that have been acknowledged by the server. These packets must not be lost due to a fault since the client may not be capable of retransmitting packets that have been acknowledged. Hence, client packets received by the server must be stored redundantly before they are acknowledged.

3.3.3. Application State

The user-level application maintains internal state that it requires in order to correctly handle requests. The *application state* is often maintained for the entire lifetime of the application, across multiple client requests and connections. Unlike connection state, where state changes only have an effect on the related connection or request, application state changes may have an effect on the handling of other future client connections and requests. In some architectures (e.g., three-tier systems), application state may be kept on a separate server host such as a database

server.

Application state changes may occur deterministically or non-deterministically. If state changes are deterministic, it may be possible to recover the state by re-executing the application after a fault. However, if state changes are non-deterministic, it may not be possible to regenerate the state by performing identical operations. Hence, it may be necessary to preserve a copy of the state whenever state changes occur. Possible approaches for state preservation are discussed further in section 3.4.

It is also possible for the service application to be stateless. In such cases, there is no application state to preserve and a reliability solution at the connection-level will suffice. However, there are subtleties that must be carefully considered. For example, a web server (application) that is providing static data (e.g., IRS forms) might be considered to be stateless since the processing of a client request does not affect the processing of any future client requests. However if relevant application state is not preserved across server failures, the system may lose client requests that have been received but for which a reply has not been sent. Hence, a web server application does maintain some state that must be preserved — received client requests and generated replies that have not been fully sent. These application-layer messages must be preserved across server failures [Aghd03b]

3.4. Approaches for State Preservation

In order to achieve client-transparent fault tolerance, a valid, up-to-date copy of the state maintained by an active server must be available to an alternate server replica. If the active server fails, the backup replica can use this state to recover active connections and resume the service with minimal interruption to the clients. There are three common approaches for preserving server state: active replication, message logging, and checkpointing. In the rest of this section, we discuss each approach and its tradeoffs in detail.

3.4.1. Active Replication

The state of a server can be replicated by having two server replicas actively perform the same exact operations on identical input. As a result, identical state changes will occur on both replicas and both servers will have an up-to-date copy of the state. The approach where two (or more) replica servers perform identical operations is called active replication.

Active replication can be used at the communication level to maintain identical connection state on both server replicas. Client packets must be multicast to both server replicas and simultaneously processed by each server. In order for a solution to be client-transparent, the multicast of client packets must be achieved without any changes to client [Aghd01, Aghd02].

Active replication can also be used to preserve the application state. An identical copy of the service application is executed on each replica [Burt02, Shen00]. In order for the replicas to reach the same state, the application input (i.e. the client requests) arriving at the replicas must also be identical. For example,

active replication of connection state can be used to ensure identical data streams reach both servers.

Active replication requires the processing to be deterministic (see Section 3.5). In practice, the processing at either the connection or application level is often non-deterministic. Thus, techniques such as synchronization of replicas must be used to handle the non-determinism. We discuss approaches for handling non-determinism in Section 3.5.

Active replication incurs a high processing overhead since all processing is performed on a replica. Hence, there is at least 100% overhead in terms of processing cycles used by the service. In Chapter 5 we show that the overall system overhead of active replication can be reduced for some applications by avoiding full replication and only selectively replicating the critical portions of an application.

3.4.2. Message Logging

The server state may also be preserved by using the message logging approach. The service input (i.e. client messages) are redundantly stored (logged) on a replica or storage node. If the server fails, logged messages are replayed and reprocessed on a new initialized alternate server, bringing the alternate server back to the pre-fault state of the failed server.

Message logging can be used at either connection [Alvi01] or application [Aghd01, Aghd02] level. To preserve connection state, all incoming client *packets* are logged on a replica server or storage node. For reliable communication, the packet is not processed until an acknowledgment is received

signaling that the packet has been safely stored. If a server fails, logged packets are replayed to an initialized alternate server configured with the same identity as the failed server, bringing the alternate server back to the pre-fault state of the failed server.

Message logging at the application level is similar to connection level logging, except that it is the user-level *application messages* that are logged to a replica or storage node. If there is a failure, the logged messages are replayed on an initialized alternate server, restoring the pre-fault server state.

There is a performance tradeoff in terms of resource usage and recovery time when choosing between the active replication and message logging approaches. Active replication requires more processing resources during fault-free operation since all processing is performed multiple times. However, since a replica has an identical copy of up-to-date state, the recovery time from failure is short. Logging does not duplicate processing during fault-free operation. Hence, it typically incurs lower processing overhead than replication. However, it suffers from longer recovery times because all logged messages must be replayed in order to reach pre-fault state.

3.4.3. Checkpointing

With checkpointing, the server state is saved in stable storage or copied to a standby backup whenever an event that changes critical state occurs, or whenever some period of time has passed since the last checkpoint. If a fault occurs, the most recent checkpointed state is recovered and processing resumes using the recovered state.

The overhead incurred by checkpointing is directly related to the frequency of checkpoints. In order to avoid any loss of service state, a checkpoint must occur with *every* state change — a very costly approach. Since frequent checkpointing of state is costly, implementations typically combine message logging with checkpointing. Messages received since the last checkpoint are logged. After a fault, the most recent checkpointed state is recovered, followed by the replaying of logged messages to reach the exact state at failure time. Hence, compared to checkpointing every state change, the overhead during normal operation is reduced. However, the recovery time is increased.

3.5. Techniques for Handling Non-determinism

Both active replication and logging schemes require servers to be deterministic — processing of identical input (i.e. request) must always result in the production of the same output (i.e. reply). However, in practice many servers are not deterministic so the use of active replication or logging may lead to an inconsistent (incorrect) state following recovery. Non-deterministic application behavior is usually caused by non-deterministic system call results. For example, calls to the *time* system call may return a different value for each replica. Hence, one approach for handling non-deterministic server behavior is to synchronize system calls made by server replicas [Burt02]. The results of system calls made by the primary server are sent to the backup replica. Backup system calls use the results obtained from the primary instead of results from its own kernel. Since applications may frequently execute a large number of system calls, this approach may suffer from a large overhead. The advantage is that knowledge of the

application internals is not required.

An alternative approach for handling non-deterministic server behavior is to identify possible sources of non-deterministic state changes and synchronize the replicas via messages at those points. Depending on the application, synchronization may be possible at coarse granularity, such as once per client request [Aghd01, Aghd02], or at finer granularity, such as at specific system calls [Zago03]. Since some real applications have relatively few non-deterministic state changes [Aghd05, Zago03], fewer synchronization messages are required compared to a scheme that synchronizes every system call, leading to lower performance overhead. The drawback is that knowledge of implementation details and modification of the application are required.

Although less frequent, non-deterministic state changes may also occur at the connection level. For example, the initial TCP sequence number for each connection is chosen non-deterministically and must be synchronized when preserving TCP connection state [Aghd02].

3.6. Error Detection

A key step of most fault tolerance mechanisms is to detect the occurrence of an error — *error detection*. Once an error is detected, *diagnosis* must be performed — the failed component is identified. In order to be able to continue with correct operation, the system must be restored to a valid state. If duplication is employed, a *failover* is performed where the failed component is replaced with a redundant replica (Section 3.7). The replica must have a copy of pre-fault state of the failed component obtained by using one of the approaches described in

Section 3.4.

The techniques that may be used for the identification of errors are dependent on the assumptions made regarding the possible behavior of failed components. These assumptions are generally referred to as the *fault model* (Section 3.2). If server hosts are fail-stop [Schn84] a heart-beat monitoring mechanism can be used to detect errors. In error-free operation, each host periodically transmits a heartbeat message. This heartbeat message is monitored, either by a replica host or by a third party monitor. If a server host fails, i.e., crashes, heartbeat messages are no longer generated. Hence, the monitor will not receive them. Missed heartbeats from a host signal that a failure has occurred. Thus, the monitor can inform interested components of the system of the failed host.

In practice, errors often have an impact at a smaller granularity than hosts. For example, an error may cause a process to crash while other processes on the host continue to operate correctly. Implementation of error detection at smaller subsystem level, e.g., heartbeat monitoring for every process, becomes more complex and costly. Hence, while we assume that individual *processes* may crash — processes are fail-stop (Section 3.2), our implementation converts process crash failures to host crash failures (Section 4.4.3).

Systems that do not make the fail-stop assumption must use more sophisticated error detection mechanisms. A heartbeat mechanism alone does not suffice since a faulty node may produce incorrect results while transmitting heartbeats. Typically, solutions use active replication of all operations, with results compared or voted in order to, respectively, detect or mask failed replicas.

3.7. Service Failover

The next step after error detection is the resumption of normal operation with a repaired system state. When the fault tolerance mechanism used is duplication, this step involves migrating the service from one replica to another. This process is called *service failover*. When an error is detected, the system must transition from standard replicated (duplex) operation to single server (simplex) mode. The identity of the service must be preserved for new and existing connections. The system must ensure that active connections and in-progress requests are processed in simplex mode. Each and every client request must be processed correctly and valid service reply messages must continue to be delivered after failover.

With many implementations of duplication, the service is unavailable during failover — no new client connections are accepted and the processing of active connections is delayed while a valid state is established. Hence, in order to mask failure from clients, the time required to perform failover operations (and error detection) must be short. This is especially critical for applications with real-time requirements, such as multimedia services or video conferencing. The failover time is not as critical to other types of applications, e.g., bulk file transfer over TCP, that are not as time sensitive. The failover time is often largely affected by the state preservation approach. For example, as discussed in Section 3.4, the message logging approach requires a large failover time since all messages received prior to the fault must be replayed. There is typically a tradeoff between the failover time and the overhead incurred during fault-free operation. The requirements of the service application often dictate the appropriate design choice.

3.8. Restoration of Fault-Tolerant Service After Failover

After a failover, it is desirable to restore the system back to its fault-tolerant configuration. A mechanism is required to integrate a new (or recovered) server into the system. The required operations are similar to the initialization of the service with one key difference: unlike initialization time, one of the servers is operating in simplex mode and actively processing client requests. The new server replica must be integrated into the system without any disruption to active connections being processed in simplex mode.

Ideally, all new connections arriving after the integration, as well as existing active connection being processed in simplex mode, would be transitioned to fault-tolerant (duplex) operation. That would allow for recovery from any new failures. The transition of active connections from simplex to duplex operation requires the applicable server-side state be transferred and replicated on the new server. This synchronization required between the active and new server can be costly and increase the transition time. To avoid this cost and complexity, our implementation differentiates between active connection at the time of integration and new connections that arrive after integration [Aghd03b]. While new connections are processed in duplex mode, active connections continue to be processed in simplex mode so that they are vulnerable to additional faults. Since web service connections are typically short-lived, the duration of this vulnerability is short.

3.9. Summary

The first step of our methodology is the identification of critical state components maintained on a server. The critical state typically includes service identity, connection state, and application state. Server failures can be transparently masked from the clients if the critical server state is preserved over failures and recovered on an alternate replica.

Once the critical state is identified, an approach must be selected for replication and preservation of this state over server failures. Three common approaches are widely used: active replication, message logging, and checkpointing. As discussed, each approach has tradeoffs involving fault-free overhead and recovery time.

One approach that we found to be effective and efficient for network services (as shown in Chapters 4 and 5) is a combination of active replication and message logging. The connection state — which changes frequently with each packet, but has a relatively low processing cost — is actively replicated on a replica host. At the application level, message logging and synchronization of non-deterministic state changes are used. Some applications may be stateless (e.g. web server) with their critical state confined to the application level messages. For these applications, message logging at the application level allows critical state to be recovered following replica failure while avoiding the processing cost that active replication would incur.

The use of message logging in order to allow a valid application state to be established following replica failure is not always the best approach. In particular, for stateful applications (e.g. video conferencing), message logging must be

accompanied by periodic checkpointing [John88]. For such applications, active replication may be the preferred approach. Active replication incurs a high processing overhead since all processing is performed on a replica. However, only the processing that affects the service state is required to be replicated. It is not necessary to replicate any processing that does not cause changes to the critical service state. Hence, in some cases an implementation can avoid full replication and reduce the overhead of the overall scheme. For example, most of the processing performed by a video conferencing server is related to the media and does not affect the critical state maintained on the server. Thus, a replicated video conferencing server can avoid the full cost of active replication by avoiding the replication of media processing. In Chapter 5 we present an efficient replicated implementation of video conferencing which eliminates the unnecessary replication of media processing and uses application-level synchronization to handle non-deterministic state changes.

In order for recovery to be initiated, errors must first be detected. If it can be assumed that system components are fail-stop, heartbeat exchanges among the components may be used to detect errors. In practice, it cannot be assumed that hosts are fail-stop. However, assuming that either processes crash (processes are fail-stop) or hosts crash (hosts are fail-stop) covers the majority of errors observed in real systems.

When an error is detected, a failover must occur. If the service identity was mapped to the failed server, a replica must take over that address in order to achieve client-transparent recovery. The preserved state of the failed server must be recovered and reused, and system operations must transition from replicated

(duplex) to simplex. Once the failover operations have completed and the system is active in simplex mode, it is desirable to restore the ability of the system to recover from the effects of additional faults. This requires the ability to integrate a new server into the system and resume replicated operation.

Chapter Four

CoRAL: A Transparent Fault-Tolerant Web Service

We have designed and implemented CoRAL [Aghd01, Aghd02, Aghd03b, Aghd03a], a fault-tolerant Web service scheme based on **C**onnection **R**eplication and **A**pplication-level **L**ogging. CoRAL recovers in-progress requests and does not require deterministic servers or changes to the clients. The server-side TCP connection state is actively replicated on a hot standby backup, allowing for fast failover. Application-level (HTTP) request and reply messages are logged to the backup and can be replayed if necessary, allowing for the handling of non-deterministic content.

In this chapter, we present our design, implementation, and experimental evaluation. Section 4.1 is an overview of the assumptions underlying our work. The system architecture and key design choices are presented in Section 4.2. A high level description of how our recovery mechanisms function is presented in Section 4.3. Based on this design, we implemented CoRAL using two approaches. Section 4.4 presents the details of our implementations: a proxy-based user-level implementation (Subsection 4.4.1) and an implementation based on kernel-level and web server modifications (Subsection 4.4.2). Techniques for optimizing CoRAL's performance under certain workloads are discussed in Section 4.5. The detailed evaluation of our scheme's overhead during failure-free operation as well as impact of failures on the service are presented in Section 4.6. The correctness of our scheme was tested by intentionally emulating (injecting) faults and monitoring

their impact. The results of our software fault injection experiments are presented in Section 4.7.

4.1. Assumptions

Fault injection experiments on other systems [Made02] have shown that most transient hardware faults either have no effect or cause a process or the entire node to crash, i.e., processes are usually *fail-stop* [Schn84]. In practice faults are infrequent. Thus, unless faults on multiple hosts are expected to be correlated, it is reasonable to assume that only one host at a time will be affected by a fault. The probability of correlated failures can be reduced by using preventive techniques such as putting machines on different power circuits [Zhan04].

Based on the above considerations, we assume that only one host at a time can be affected by a fault and that the impact of the fault can be to either crash a process or crash the entire host. As discussed later in Subsection 4.4.3, our implementation converts process crash failures to host crash failures. Thus, most of the discussion is focused on host crash failures (*fail-stop* hosts).

We assume that the local area network connecting the two servers as well as the Internet connection between the client and the server LAN will not suffer any *permanent* faults. The primary and backup hosts are connected on the same IP subnet. In practice, the reliability of the network connection to that subnet can be enhanced using multiple routers running protocols such as the Virtual Router Redundancy Protocol [Knig98]. This can prevent the local LAN router from being a critical single point of failure. Our scheme does not currently deal with the possibility that the two hosts become disconnected from each other while both

maintaining their connection to the Internet. Forwarding heartbeats through multiple “third parties” could be used to detect this situation and continue normal operation in a degraded mode or intentionally terminate one of the servers. We assume that the primary and backup are physically close, so that communication between the two servers is much faster than communication with the client.

4.2. System Architecture

In order to provide client-transparent fault-tolerant web service, a fault-free client must receive a valid reply for every request that is viewed by the client as having been delivered. Both the request and the reply may consist of multiple TCP packets. Once a request TCP packet has been acknowledged by a server, it must not be lost. All reply TCP packets sent to the client must form consistent, correct replies to prior requests.

The basic idea behind CoRAL is to use a *combination* of active replication and logging. There is a *primary* server and a *backup* server. At the connection level, the server side TCP state is actively replicated on the primary and backup. However, the standby backup server [Borg83] logs HTTP requests and replies and does not *process* requests unless the primary server fails. Clients communicate with the service using a single server address, composed of an IP address and TCP port number, henceforth called the *advertised address*. At the TCP/IP level, all messages sent by clients have the advertised address as the destination and all messages received by clients have the advertised address as the source address. The primary and backup hosts are connected on the same IP subnet, which is also the subnet of the advertised address.

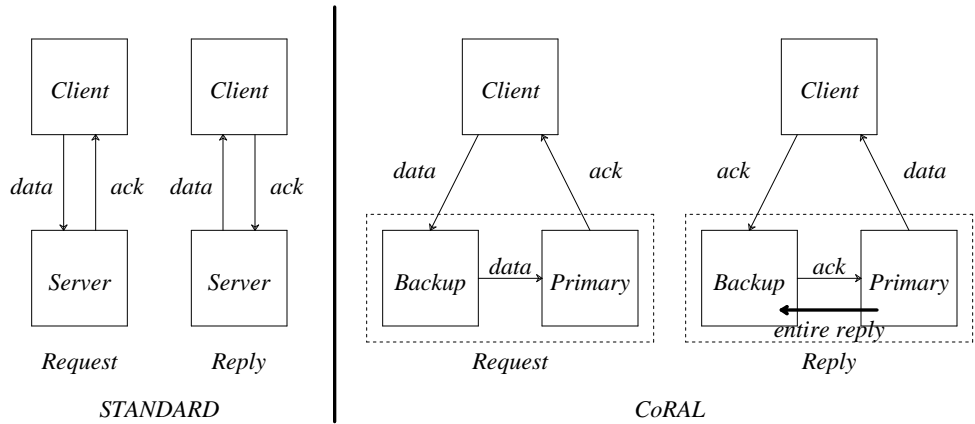


Figure 4.1: Message paths for a standard unreplicated server and a hot standby replication scheme. Replicated servers appear as a single entity to clients. The reply is generated by the primary and reliably sent to the backup before being sent to the client.

Figure 4.1 shows the building blocks and message paths in CoRAL. The primary and backup receive and process every TCP packet. In fault-free operation, the advertised address is mapped to the backup. Hence, the backup server receives all packets with the advertised address as their destination. Upon receipt of a packet, the backup server forwards a copy of the packet to the primary server by changing the destination address of the packet. The packet's source address remains the client's address. Thus, these packets appear to the primary server as though they were sent directly by the client. The primary server generates and sends the TCP acknowledgment packets to the client, using the advertised address as the source address. This scheme ensures that the backup has a copy of each request before it is available to the primary.

At the application level, HTTP request and reply messages are logged. The backup logs each request while the primary processes the request and generates a reply. Once a reply is generated by the primary, a complete copy is reliably sent to

the backup before any reply is sent to the client. The reliability of this transmission is assured using an explicit acknowledgment from the backup upon receipt of the entire reply. Upon receiving this acknowledgment, the primary sends the reply to the client. If the primary fails *before* starting to transmit the reply to the client, the backup transmits its copy. If the primary fails while sending the reply to the client, the error handling mechanisms of TCP are used to ensure that the unsent part of the reply will be sent by the backup. If the primary fails before logging the reply, the backup processes its copy of the request, generates a reply, and sends it to the client. As with acknowledgement packets, the advertised address is used as the source address of reply packets sent to the client. Upon the receipt of the reply data packets, the client sends TCP acknowledgment to the source of the replies, i.e., the advertised address. The backup server receives these acknowledgments and forwards them to the primary server in the same manner as client data packets.

The key to the scheme described above is that the backup server obtains *every* TCP packet (data or acknowledgment) from the client *before* the primary server. Thus, the only way the primary obtains a packet from the client is if the backup already has a copy of the packet. Replies (TCP data packets) generated by the primary server are logged to the backup before they are sent to the client. Since all the acknowledgments from the client arrive at the backup before they arrive at the primary, the backup can easily determine which replies or portions of replies it needs to send to the client if the primary fails.

While our implementation does not include the communication between the front-end and back-end servers, this can be done as a mirror image of the communication between the client and front-end servers. Furthermore, since the

transparency of the fault tolerance scheme is not critical between the web server and back-end servers, simpler and less costly schemes are possible for this section. For example, the front-end servers may include a transaction ID with each request to the back-end. If a request is retransmitted, it will include the transaction ID and the back-end can use that to avoid performing the transaction multiple times [Orac99a].

4.3. Failure Recovery

In the event of a server failure, the surviving server replica must take over and continue providing service to the clients, including handling of in-progress requests — requests that were being processed at failure time. If the backup server fails, the primary server takes over the advertised address and starts operating in simplex mode. If the primary server fails, the backup begins processing HTTP requests and sending replies in simplex mode. Table 4.1 summarizes the mechanisms used to recover from server failures that occur during different phases of handling HTTP requests and replies.

Our system can tolerate single server failure as well as transient communication faults that lead to lost, duplicated, or corrupted packets. Packet loss, corruption, or duplication are handled by TCP's error handling mechanisms. Table 4.2 summarizes possible communication errors and the mechanisms used to recover from them.

failed server	HTTP message processing phase	recovery mechanism
backup	after backup receives an incomplete client HTTP message	some client TCP data packets of the HTTP message are not acknowledged since they were not received by primary. primary takes over advertised address, client retransmits and/or transmits any unacknowledged TCP packets of the message, primary receives and processes entire message
backup	after backup receives a complete client HTTP message but before all of it is relayed to primary	primary takes over advertised address, since some client TCP data packets are unacknowledged, they are retransmitted by the client, primary receives and processes entire message
backup	after backup forwards a complete client HTTP message to primary	all client TCP data packets are properly acknowledged by primary, primary handles message, primary takes over advertised address and handles future requests
backup	no HTTP request or reply in progress	primary maps the advertised address to itself and starts operating in simplex mode handling all future requests
primary	after primary receives an incomplete or complete client HTTP message but before it can acknowledge the last client TCP data packet of the message	unacknowledged client TCP data packets are retransmitted, backup takes over and starts operating in simplex mode, backup acknowledges all unacknowledged client TCP data packets and processes message
primary	after primary receives and acknowledges a complete HTTP request but before it sends a full copy of the HTTP reply to backup	backup takes over and starts operating in simplex mode, backup starts processing its copy of pending requests for which replies have not been transmitted, backup generates and sends HTTP reply to client
primary	after primary sends a complete copy of an HTTP reply to backup but before transmitting some of the reply TCP data packets to client	backup takes over and starts operating in simplex mode, backup starts processing its stored copies of HTTP replies which have not been completely acknowledged by client, backup transmits missing TCP data packets of HTTP reply to client
primary	no HTTP request or reply in progress	backup takes over and starts operating in simplex mode handling all future requests

Table 4.1: Recovery from server failures that occur during different phases of handling HTTP requests and replies.

4.4. Implementation

The scheme described in Section 4.2 can be implemented in several ways.

packet lost/corrupted	detection and recovery mechanism
client data: client → backup	primary doesn't receive packet, no ack to client, client retransmits
client data: backup → primary	primary doesn't receive packet, no ack to client, client retransmits, backup ignores retransmitted packet as duplicate but still relays it to primary
server ack: primary → client	client doesn't receive ack and thus retransmits data packet, primary (and backup) ignore retransmitted data packet as duplicate and primary retransmits ack
server data: primary → backup	no ack to primary, primary retransmits server data packet to backup
server ack: backup → primary	no ack to primary, primary retransmits server data packet to backup, backup ignores data packet as duplicate and retransmits ack to primary
server data: primary → client	no ack to primary, primary retransmits server data packet to client
client ack: client → backup	primary doesn't receive ack, primary retransmits server data packet, client ignores retransmitted packet as duplicate but still retransmits ack
client ack: backup → primary	primary doesn't receive ack, primary retransmits server data packet, client ignores retransmitted packet as duplicate but still sends ack, backup ignores duplicate ack but relays it to primary

Table 4.2: Communication errors and the mechanism used to recover from them. All actions except relays by the backup are a direct result of using TCP.

We have implemented the scheme using two approaches. The first approach is based on user-level proxies [Aghd01]. It does not require any kernel modifications and requires minimal (if any) changes to the web server software. The second approach consists of a combination of kernel modifications and modifications to the user-level web server using their respective module mechanisms [Aghd02]. The proxy-based implementation is simpler and potentially more portable than an implementation that requires kernel modification. However, it also incurs a higher performance overhead (e.g., multiple user/kernel context switches for processing of each packet) and requires stricter assumptions (e.g., multiple proxies being

operational or fail-stop as a single entity). The approach based on a combination of web server and kernel-level modifications is more efficient but it is more difficult to implement. As discussed in Section 4.4.2, our modular approach simplifies the porting of this implementation to other kernels or web servers. As an alternative to our implementations, it is also possible to implement the scheme entirely in the kernel using a kernel-level web server [RedH01]. A kernel-only implementation may possibly reduce the scheme's overhead. However it is generally desirable to minimize the complexity of the kernel [Blac92, Golu90], and such an approach may lead to the degradation of the overall system performance and reduce the kernel's robustness.

4.4.1. A Proxy-Based User-Level Implementation

The tasks of the scheme described in Section 4.2 can be divided into two categories: modification of TCP packet headers and operations performed at the HTTP message granularity. In our proxy-based implementation, each of these categories of tasks is implemented by a separate process, henceforth referred to as *proxy*. This leads to a simple modular design and also has the potential to facilitate pipelining (on a multiprocessor) of the handling of requests and replies. The proxy that performs operations at the granularity of IP packets is called the *raw proxy* since it uses the socket interface in the RAW mode to gain access to the packet headers. The proxy that performs operations at the granularity of HTTP messages is called the *TCP proxy* since it uses the socket interface in the STREAM (TCP) mode. The TCP proxy sends and receives complete HTTP messages and buffers HTTP requests and replies for fault recovery.

Our scheme requires processing of IP packets in ways that do not match standard kernel-level implementations of TCP. This processing could be accomplished by modifying the kernel, however avoiding kernel modifications is advantageous for overall system reliability and for simplicity and portability. Using the “raw socket” interface with Sun Microsystem’s Solaris operating system, it is possible to avoid most kernel processing of packets and perform non-standard modifications of packet headers at user-level. However, once the packets reach the user-level they must be processed in a manner consistent with standard TCP protocol. Hence, the use of the raw socket interface could lead to a requirement for a full user-level TCP implementation on the servers for communication with the client. Such a user-level TCP implementation could be integrated with the web server code and modified to provide all the required functionality. User-level TCP implementations are not easily available and often lack the robustness of kernel-level implementations. On the other hand, kernel-level TCP implementations are relatively robust since they are critical to the operation of the system and subject to extensive testing and use by developers and users. Thus, we chose to avoid using an user-level TCP implementation and instead use a second-tier of proxies (TCP proxies) described below. Our user-level proxy based implementation uses proxies to avoid both kernel modifications and user-level TCP implementations while minimizing changes to the servers; all at the cost of additional overhead.

The system is comprised of a raw proxy and a TCP proxy for the primary server as well as a raw proxy and a TCP proxy for the backup server. We refer to the triple of raw proxy, TCP proxy, and server as a *cluster*. The functionality of

the “primary server,” as described in Section 4.2, is implemented by the primary *cluster*. The functionality of the “backup server,” as described in Section 4.2, is implemented by the backup *cluster*. We assume that each of the clusters is either operational or fail-stop as a single entity. Using fail-stop hosts [Schn84], this assumption can be enforced using heartbeats from each host running any cluster component coupled with intentional termination of a cluster if any of its components fail. The overall structure of the system is shown in Figure 4.2.

The backup TCP proxy logs HTTP requests from the client (arriving via the backup raw proxy). The primary server gets each request from the primary TCP proxy and sends the HTTP reply back to the primary TCP proxy. From the primary TCP proxy the entire reply is sent to the backup TCP proxy. The backup TCP proxy must then match this reply with the corresponding HTTP client request received previously. Each client HTTP request can be uniquely identified by the client address and a request sequence number. However, since the TCP proxies communicate with the raw proxies and not directly with the client, the TCP proxy does not automatically have the client address. To solve this problem the raw proxies add the client address to the HTTP requests before forwarding them to the TCP proxies. The primary TCP proxy removes the client address from the HTTP request before it is sent to the primary server. These are items 3 and 6 in the description of TCP level events below (Figure 4.2).

In order to explain the operation of our system, we provide a step-by-step explanation of the progress of requests and replies when the system is operating in its normal fault-tolerant (duplex) mode. The numbers of the steps correspond to the labels in Figure 4.2.

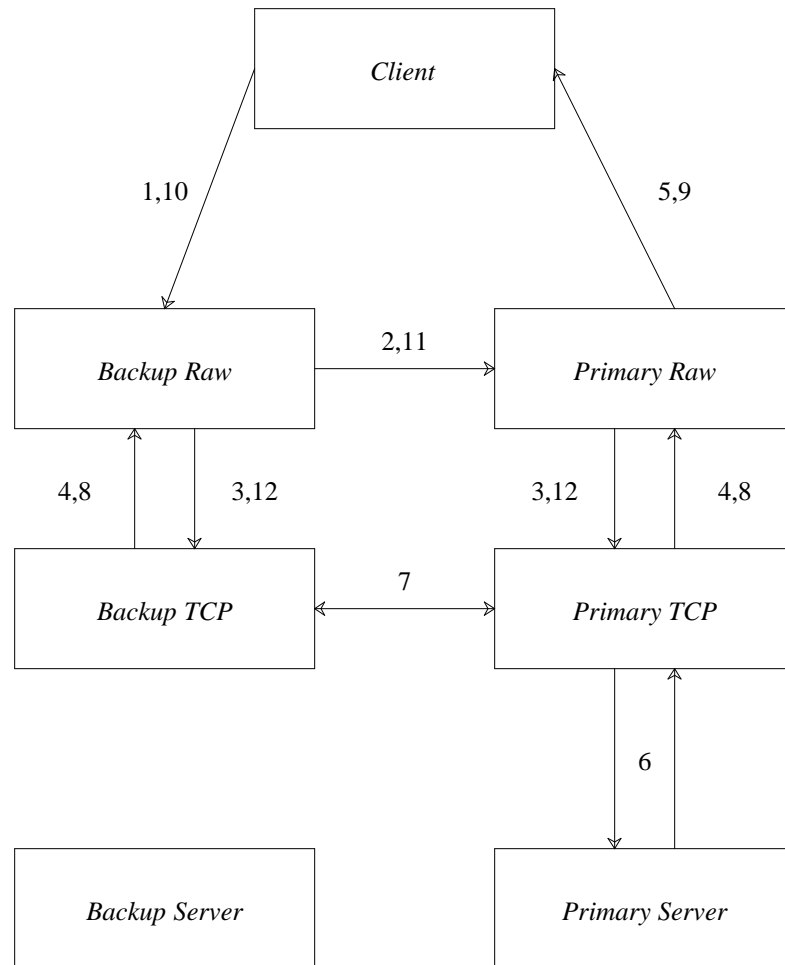


Figure 4.2: System structure for client-transparent fault tolerance. The connections shown are the TCP/IP packet routes for normal fault-tolerant (duplex mode) operation.

1. The client sends a data packet to the advertised address, which is mapped to the backup raw proxy.
2. The backup raw proxy reads the packet, makes a copy, and changes its destination address to the primary raw proxy (the source address remains the address of the client). The packet is then sent to the primary raw proxy.

3. Each raw proxy changes the source address of the client's packet to its own address. If this is the first data packet from the particular client address, the raw proxy also appends the client IP address and TCP port number to the TCP data and makes appropriate changes to the packet's TCP sequence number. The modified packet is then sent to each raw proxy's respective TCP proxy.
4. The TCP proxies receive the packets sent to them by the raw proxies. The OS kernels on which the TCP proxies are running send TCP acknowledgment packets back to the raw proxies.
5. The raw proxies receive the TCP acknowledgment packets from the TCP proxies. The primary raw proxy changes the source address of the packet to the advertised address and the destination address to the client address. It also modifies the TCP acknowledgment number to account for the extra bytes that are sent to the TCP proxy at step 3 since these extra bytes are acknowledged by the TCP proxies. The primary raw proxy then sends this packet to the client. In duplex mode, the backup raw proxy drops the packet.
6. When enough TCP data packets have arrived at the TCP proxy to compose an entire HTTP request, the TCP proxies remove the extra bytes (client address) that are placed in the request by the raw proxies (step 3). The backup TCP proxy logs the received request. The primary TCP proxy sends the HTTP request to the primary server via a TCP connection. If all goes well, the primary TCP proxy then receives the HTTP reply from the primary server. Note that this entire step can occur simultaneously with step 5.
7. The primary TCP proxy appends the client address that it removed from the HTTP request in step 6 to the HTTP reply and sends the reply to the backup

TCP proxy via a TCP connection. The backup TCP proxy receives the HTTP reply and uses the embedded client address to match the reply with the HTTP request that it received in step 3.

8. The TCP proxies send the HTTP reply to their respective raw proxies. The HTTP replies may be broken up (by the OS kernel) into several TCP packets.
9. The raw proxies change the source address to the advertised address, the destination address to the client's address, and adjust the TCP sequence number. Normally, the backup raw proxy discards the packet. The primary raw proxy sends the TCP packets to the client.
10. The client receives the TCP data packets (HTTP reply) and sends acknowledgment packets to the source address of the data packets, i.e., the advertised address.
11. Similar to step 2, the backup raw proxy receives the acknowledgment packet, changes its destination address to the primary raw proxy (maintaining the client as the source address) and sends the packet.
12. The raw proxies change the TCP acknowledgment numbers to match the TCP sequence numbers used by the TCP proxies (reverse of step 9) and send the packets to their respective TCP proxies.

The TCP sequence number adjustments in steps 9 and 12 are necessary in order to ensure that both the primary and backup are using the same sequence numbers, allowing a transparent switch to simplex mode operation in case of failure. Each TCP proxy is free to choose any sequence number when initializing a TCP connection. The backup raw proxy selects the actual initial TCP sequence number that is used by the servers when communicating with the client. After the

selection, the backup raw proxy sends this initial sequence number to the primary raw proxy, ensuring that both raw proxies will use the same sequence number space when communicating with the client (see kernel module section for more detail). Each raw proxy calculates the offset of the actual TCP sequence number and the TCP sequence number generated by the corresponding TCP proxy. This offset is then used to modify the TCP sequence number of packets being sent to the client for the lifetime of the connection.

Since only the primary raw proxy sends packets to the client (step 9), a problem can arise if the backup cluster falls behind the primary and as a result receives TCP acknowledgments for packets that it has not yet sent. To solve this problem the backup raw proxy keeps track of TCP acknowledgment packets that it receives. Whenever it receives a TCP data packet from the backup TCP proxy that contains a TCP sequence number that has already been acknowledged by the client, the backup raw proxy generates a “fake” acknowledgment packet with the client address as the source and sends it to the backup TCP proxy. This allows “old” data at the backup TCP proxy to be acknowledged.

4.4.2. An Implementation Based on Kernel-Level and Web Server Modifications

Although the user-level proxy implementation presented in Section 4.4.1 is simpler and potentially more portable, its associated performance overheads proved too costly for practical settings. In particular, there is a difference of almost a factor of five in the required CPU cycles per request between the user-level proxy approach and the approach that includes kernel-level modifications (see

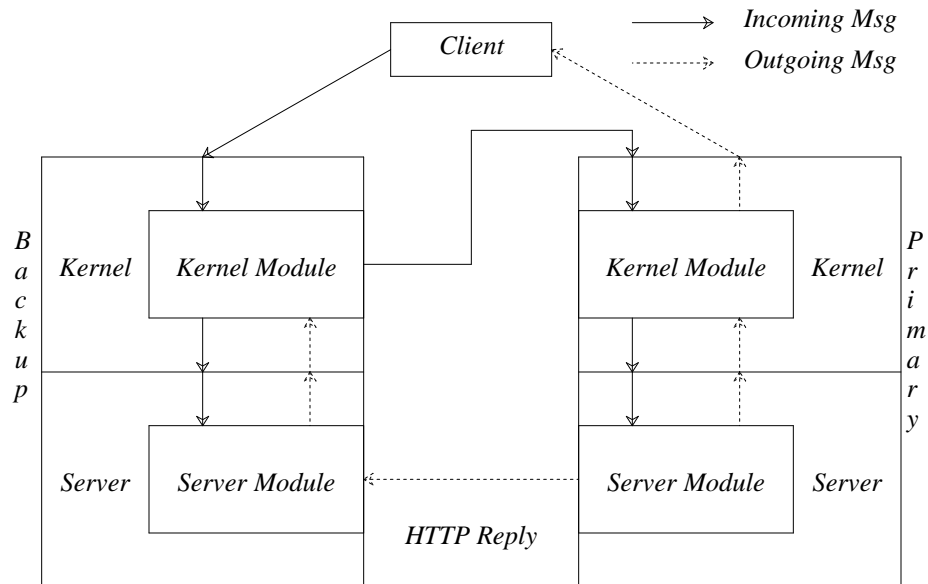


Figure 4.3: Implementation: kernel and web server modules are used to provide the necessary mechanisms for replication. Message paths are shown.

Section 4.6.4). Hence, in practice a more efficient implementation is necessary.

As discussed previously, the tasks of the scheme described in Section 4.2 can be divided into two categories: packet header modifications and operations performed at the HTTP message granularity. In our implementation based on kernel-level and web server modifications presented in this section, each of these categories of tasks is implemented by a separate module (Figure 4.3). Kernel modifications, implemented in a Linux loadable kernel module, perform TCP/IP packet operations. HTTP message operations are performed in the Web servers and are implemented in an Apache Web server module. In the rest of this section, we present the details of our kernel module, Web server module, and service failover.

4.4.2.1. Kernel Modifications

The kernel modifications implement the client-transparent atomic multicast mechanism between the client and the primary/backup server pair. In addition, modifications facilitate the transmission of outgoing messages from the server pair to the client such that the backup can continue the transmission of replies seamlessly if the primary fails. The kernel modifications modify the operation of the kernel as follows: 1) a copy of incoming client packets arriving at the backup are forwarded to the primary, 2) outgoing (from the primary to the client) packets' headers are rewritten to use the advertised address as source, 3) outgoing packets generated at the backup are dropped during normal fault-free operation, and 4) server-side initial TCP sequence numbers are synchronized between the primary and backup, and packet headers are rewritten to ensure the use of a consistent sequence number space. The details of these modifications are discussed in the rest of this subsection.

As in the user-level proxy implementation, the advertised address of the service known to clients is mapped to the backup server, so the backup will receive the client packets. After an incoming packet goes through the standard kernel operations such as checksum checking, the backup's modified kernel forwards a copy of the packet to the primary. The backup's kernel then continues the standard processing of the packet, as does the primary's kernel with the forwarded packet.

The primary server sends the outgoing packets to the clients. Again as in the user-level implementation, such packets must be presented to the client with the advertised address as the source address. Hence, the primary's kernel modifications change the source address of outgoing packets to the advertised

address. On the backup, the kernel processes the outgoing packet and updates the kernel's TCP state, but the modified kernel intercepts and drops the packet when it reaches the device queue. TCP acknowledgments for outgoing packets are, of course, incoming packets and they are multicast to the primary and backup as above.

The key to our multicast implementation is that when the primary receives a packet, it is assured that the backup has an identical copy of the packet. The backup forwards a packet only *after* the packet passes through the kernel code where a packet may be dropped due to a detected error (e.g. checksum) or heavy load. If a forwarded packet is lost while enroute to the primary, the client does not receive an acknowledgment and thus retransmits the packet. This is because only the primary's TCP acknowledgments reach the client. TCP acknowledgments generated by the backup are dropped by the backup's modified kernel.

Another issue that the kernel modifications address is the selection of the initial TCP sequence number. To achieve client transparency at the TCP level, both servers must choose identical sequence numbers during connection establishment. For security reasons, most TCP stack implementations select initial sequence numbers using a random component. Hence, our scheme requires this value to be passed from one server to the other. We avoided any extraneous message passing between the servers by passing the initial sequence number in the unused ack field of TCP SYN packets. Upon the receipt of a SYN packet from the client, the backup server calculates an initial sequence number using the kernel's standard secure functions. The chosen sequence number is then placed in the ack field of the SYN packet before it is forwarded to the primary. A standard kernel

would ignore the ack field of a SYN packet since the ACK flag in a standard TCP SYN packet is not set. However, our modified kernel at the primary server expects this value and uses it as its initial sequence number.

The kernel-level implementation does not require any modification to the data portion of TCP packets. As described previously, the raw proxies in our proxy based user-level implementation modify the data portion of TCP packets in order to add a connection identifier (i.e. the client address) to each request. Our implementation choice of using two levels of proxies caused this requirement. Since the raw proxies mediate between the clients and TCP proxies, the source addresses of packets that arrive at the TCP proxies are the addresses of the raw proxies and not the client. Hence, the TCP proxies can not use standard socket calls to obtain information regarding the clients. The raw proxies must provide any desired information about the clients to the TCP proxies. With the kernel-level implementation, however, copies of client TCP packets reach both the primary and backup kernels. As a result, the user-level server processes can use standard socket calls to obtain client addresses, thus avoiding the need to modify the data portion of any TCP packets.

Our kernel modifications are implemented in the form of a loadable Linux kernel module. The module is loaded onto each server replica at initialization time. In addition to the kernel module, we modified the kernel itself and added a few function “hooks” (i.e., entry points) where our module code is called. The use of a kernel module allows for isolation of our implementation code from the rest of the kernel source. It also simplifies debugging and modifications of the code since only the module (and not the entire kernel) must be recompiled. The

total size of our kernel modification are roughly 5000 lines of code for the kernel module with small changes (less than 100 lines) to the kernel source as mentioned above.

4.4.2.2. Modifications to the Server Application

The server modifications are used to handle the parts of the scheme that deal with messages at the HTTP level. The operations of a standard off-the-shelf web server are modified to: 1) log arriving requests at the backup, 2) implement a user-level reliable reply logging mechanism where each generated reply on the primary is first logged to the backup before being sent to the client, and 3) garbage collect the logged requests on the backup when the matching reply arrives from the client. We implemented the server modification as an Apache module. The Apache web server [Apac98] implementation includes several entry points where a module may interface. Our Apache module acts as a handler [Ste99] and generates the replies that are sent to the clients. Our module is composed of roughly 6500 lines of C code. It implements worker, mux, and demux processes. The Apache code itself (approximately 100,000 lines of code) was not modified. The details are described in the rest of this subsection.

4.4.2.2.1. Worker Processes

A standard Apache web server consists of several processes handling client requests. We refer to these standard processes as worker processes. In addition to the standard handling of requests, in our scheme the worker processes also communicate with the mux/demux processes described in the next subsection.

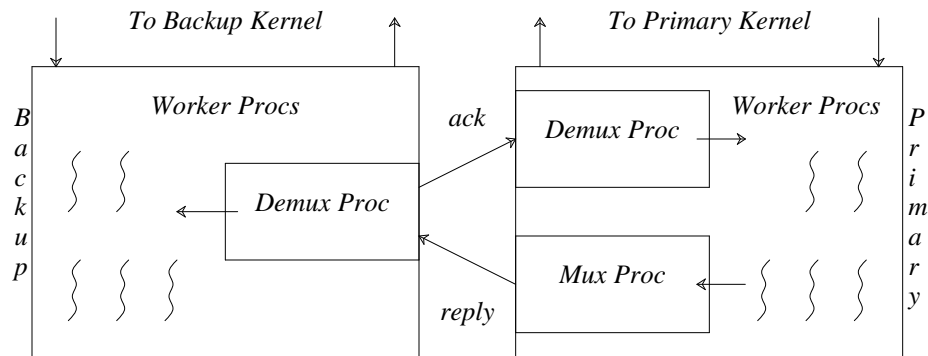


Figure 4.4: Server Structure: The mux/demux processes are used to reliably transmit a copy of the replies to the backup before they are sent to clients. The server module implements these processes and the necessary changes to the standard worker processes.

The primary worker processes receive the client requests, perform parsing and other standard operations, and then generate the replies. Other than a few new bookkeeping operations, these operations are exactly what is done in a standard web server. After generating the reply, instead of sending the reply directly to the client, a primary worker process passes the generated reply to the primary mux process so that it can be sent to the backup. The primary worker process then waits for an indication from the primary demux process that an acknowledgment has been received from the backup, signaling that it can now send the reply to the client.

The backup worker processes perform the standard operations for receiving a request, but do not generate the reply. Upon receiving a request and performing the standard operations, the worker process just waits for a reply from the backup demux process. This is the reply that is produced by a primary worker process for the same client request.

4.4.2.2.2. Mux/Demux Processes

If each server replica contained only a single worker process, the reply logging step of our scheme could be implemented simply with a direct communication between the two worker processes. Upon generation of each reply, the worker process on the primary would send a copy of the reply to the backup worker process, wait for a user-level acknowledgement, and then send the reply to the client. Practical servers however, typically contain multiple worker processes or threads. Hence, a mechanism is required to link the worker process from each replica that is handling the same request.

One possible approach would be to pair worker processes (one on each replica) statically at server initialization time. This approach would be simple, however, it requires the paired processes to always receive and process identical requests. In practice (e.g. Apache web server), the handoff of new client connections to a worker process is typically done by the kernel (via listen and accept calls) and is not deterministic. Thus, we can not ensure at the user-level that the both processes in the pair will always receive identical requests. Hence, an alternative approach is required.

An alternative approach is to have connections between each and every possible combination of process pairs on both replicas. Obviously a large initialization and resource overhead is incurred. In addition, with each request a primary process must identify which remote process is handling the same request.

Our approach avoids the overhead and implementation complexity by using dedicated processes on each replica for reply logging. There are two processes added to the primary (a “mux” and a “demux”) as well as a “demux” processes

added to the backup. The mux/demux processes ensure that a copy of each reply generated by the primary is sent to and received by the backup before the transmission of the reply to the client starts. The mux/demux processes communicate with each other over a TCP connection, and use semaphores and shared memory to communicate with worker processes on the same host (Figure 4.4).

The primary mux process receives the replies generated by primary worker processes and sends them to the backup on a TCP/IP connection that is established at startup. A connection identifier (client's IP address and TCP port number) is added as a custom header to each reply message so that the backup can identify the worker process with the matching request. The main reason for existence of this process is that there is no easy way for multiple processes to share a single connection/socket descriptor for sending. For a multi-threaded server implementation, this process would not be necessary as all threads of a process can share and use the same socket descriptor.

The backup demux process receives replies from the primary and sends an explicit (i.e. user-level) acknowledgment back to the primary for each reply. The reply's connection identifier is included in the acknowledgment message. The backup demux process then examines the custom header of each reply and hands off the reply body to the appropriate backup worker process.

The primary demux process receives acknowledgment messages from the backup and signals the appropriate worker process that its reply has been logged by the backup and that it may proceed with sending of the reply to the client. Again, the connection identifier is used to match the acknowledgments with the

appropriate worker process.

4.4.3. Converting Process Crashes to Host Crashes

As discussed in Section 4.1, our work is based on the assumption that only one host at a time can be affected by a fault and that the impact of the fault can be to either crash a process (fail-stop processes) *or* crash the entire host. However, the fault tolerance mechanisms can be simplified if it can be assumed that the only possible impact of a fault is for the host to crash (fail-stop hosts). In particular, as discussed below, there are two key complications with fail-stop processes as opposed to fail-stop hosts: 1) as a result of a fault on a server, the client may close the connection and thus the fault tolerance mechanism is no longer client-transparent, and 2) error detection may be more complex. In order to avoid these complications, our implementation converts process crashes to host crashes.

With a standard UNIX/Linux kernel, when a server process crashes, the kernel closes all open network connections of that process. As a result, the TCP implementation in the kernel generates either an RST or a FIN packet, depending on the state of the connection. Upon receiving this packet, the client on the other side of the connection will close the connection. Once the TCP connection is closed, re-establishing the connection would have to involve special action by the client, thus violating the requirement of client-transparent fault tolerance. Hence, our implementation must detect process crashes and prevent the transmission of these RST and FIN packets. This requires the implementation to distinguish between TCP RST or FIN packets generated due to a process crash, which must be discarded, and those RST or FIN packets generated during normal operation, which

must be allowed to reach the client.

Only minor changes in the kernel were required in order to discard the RST or FIN packets when a process crashes [Aghd05]. These changes identify process crashes by setting a special flag when the process performs an explicit exit call. During process termination cleanup, if this flag is set, any TCP packets generated for any of the open sockets of the process are transmitted normally. However, if the flag is not set, thus indicating that the process terminated abnormally, all outgoing packets for sockets of this process are discarded.

As described in Subsection 4.4.2.2, the server implementation consists of multiple processes on each host. If one of these processes crashes, e.g., the mux or demux processes, a complex fault tolerance scheme would be required in order to identify which process crashed and take recovery actions that would allow other processes on the host to continue to run correctly. Instead, our implementation responds to any process crash by killing *all* the service-related processes on the host. On each server host there is a process that generates periodic heartbeats and sends them to the other member of the (primary, backup) host pair. If the heartbeat monitor process on a host detects missing heartbeats from its partner host, it takes recovery actions that are appropriate for a host crash (Subsection 4.6.3). The heartbeat generator process is among the processes that are killed if any of the processes on the host crashes.

All the service-related processes on each server host are descendants of a single ancestor, henceforth referred to as the *top process*. Except for the top process, every other process is forked from another process and is thus a “child” of some other process. Whenever a child process crashes or terminates normally,

standard UNIX/Linux functionality is that the parent is notified via a SIGCHLD signal. The kernel passes to the SIGCHLD signal handler a flag that indicates whether the process exited normally or terminated abnormally. In our implementation, every process includes a handler for the SIGCHLD signal. If the child process terminated abnormally, the parent process kills all other service-related processes, including itself and the heartbeat generator. This ensures that the other member of the server pair will eventually detect a fault (missing heartbeats) and take over, as described in the next subsection. The top process on the host cannot be monitored in this fashion. Instead, the heartbeat generator explicitly checks that its parent process (which is the top process) is still alive before sending each heartbeat. If the parent process is not alive, the heartbeat generator kills all other service-related processes, including itself.

4.4.4. Failover

We assume failures to be fail-stop [Schn84]. For fault detection, heartbeat messages are exchanged between servers. A process in the user-level server modules of each server periodically sends sequenced UDP packets to its counterpart. Consecutive missed heartbeats signal that a fault has been detected.

When a fault is detected, the system must transition from standard replicated (duplex) operation to single server (simplex) mode (Figure 4.5) [Aghd03b]. The identity of the service (i.e., advertised address) is preserved for new and existing connections. Existing active connections are migrated to simplex mode. For each received HTTP request, the system ensures that the complete HTTP reply message is delivered to the client.

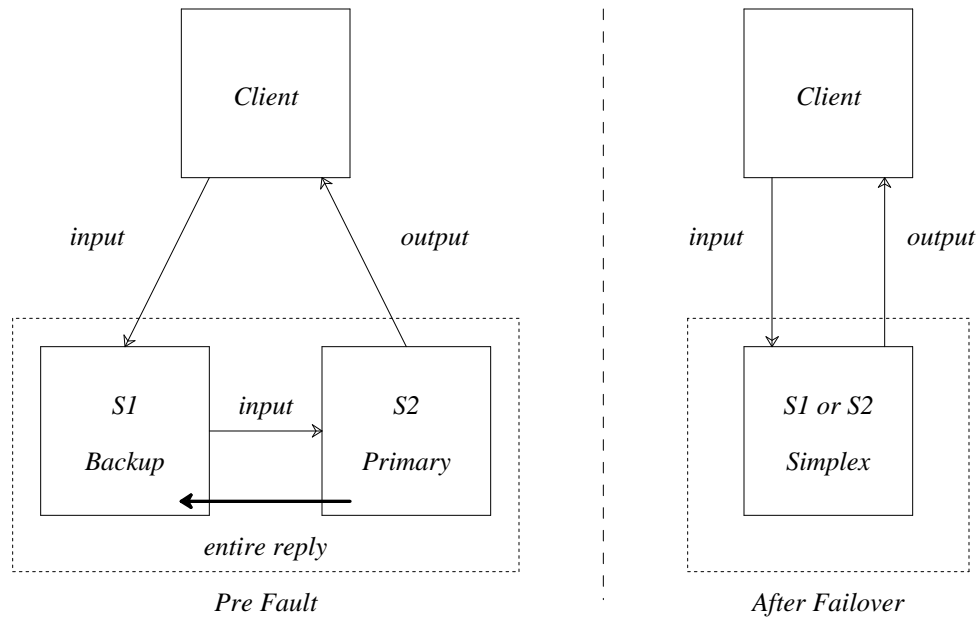


Figure 4.5: In normal (pre-fault) operation, client input is sent to the backup and then forwarded to the primary. The primary generates the output, logs it to the backup, and sends it to the client. After a fault, the surviving node takes over the identity of the failed node and operates in simplex mode.

As previously mentioned, the advertised address is mapped to the backup server in fault-free operation. The service address used by the clients must continue to be available following a backup server failure. Hence, the remaining member of the duplex pair, the primary, has to take over that IP address. This takeover can be implemented using a Linux *ioctl* that establishes an additional IP address alias for the network interface. However, as discussed later in this section, this functionality is included in a new system call that we implemented.

In a standard modern networking setup, the servers are connected to a switched LAN and all packets from remote clients pass through multiple routers on their way to the server. Following IP address takeover, the local router must be

informed that a new host (MAC address) should now receive packets sent to the “migrated” IP address. We use gratuitous ARP [Plum82] to accomplish this task. Specifically, ARP reply packets are sent to hosts (including the router) on the same IP subnet without waiting for explicit ARP requests. Once the router receives an ARP reply packet, its ARP cache is updated, causing it to route packets that are sent to the service address to the surviving server. Gratuitous ARP is not reliable since the ARP reply packets may be lost. Hence, we added a level of reliability by pinging a known host outside the server subnet. If a ping reply is received, it implies that the router’s ARP cache has been updated. If a reply is not received within a timeout interval, the gratuitous ARP reply packets are retransmitted. Only a single outside host is used in our implementation. However, this approach can be trivially extended to ping multiple outside hosts to avoid the possibility of the outside host becoming a single point of failure. A ping reply from any outside host would indicate a successful router ARP cache update.

Since faults are detected by the user-level heartbeat processes in the server module, the kernel module must be informed in order to transition active connections from duplex to simplex processing mode. Hence, we have implemented a system call which allows a user-level process to notify the kernel of a fault detection. The kernel module’s transition from duplex to simplex mode is simple. If the primary fails, the backup kernel module no longer forwards the incoming client packets to a primary. Also, outgoing packets are sent to the client instead of being discarded. As a result, unacknowledged portions of any logged replies will reach the clients. If the backup fails, the primary kernel module takes over the advertised address and incoming packets are received directly from the

clients instead of being forwarded from the backup, but this change is not noticeable to the server.

At the user-level, the heartbeat process must notify all other user-level worker and mux/demux processes of a detected fault. The user-level notification is implemented by setting a global flag in shared memory. Each server module process checks this flag when it receives a client request, and determines whether the system is in duplex or simplex mode. At failure time, some of the server module processes may be waiting for an event. For example, a backup worker process may be waiting for a reply, or a primary worker process may be waiting for acknowledgement that its reply has been logged. When a fault occurs, these waiting processes must be notified to no longer wait, since the events they are waiting for will not occur in post-fault (simplex) operation. The process that detects the fault performs these notifications using the same mechanism normally used by the demux process.

After a failover and transition to simplex mode, new requests are processed in simplex mode and replies are sent directly to the client. In-progress requests logged at the Backup, for which a reply was never logged (by the failed Primary), are similarly processed in simplex mode.

4.4.5. Restoration of Fault-Tolerant Service After Failover

After recovery from a fault, it is desirable to restore the system back to its replicated configuration so that other faults can be tolerated. Our implementation allows for the integration of a new server into the system without any disruption to active connections. No extra resources other than a process listening for a

connection are used while operating in simplex mode. After a failover to simplex mode, a server module process listens for a new server that may want to join the system. The new server trying to join the system is initialized with all the required modules and processes for duplex operation before contacting the active simplex server. The new server connects to the waiting simplex server process and the two exchange identity and configuration information. At this point, the active simplex server begins the transition to duplex mode. At the user-level, processes required for the logging of replies in duplex mode are spawned and initialized. Their initialization includes setting up the required TCP connections with the new server. Once all processes are initialized, the active simplex server invokes our system call, notifying the kernel module to also transition to duplex mode. Packets for new connections received after the transition are processed in duplex mode.

Our implementation does not transition the existing client connections being processed in simplex mode to duplex mode. Hence, a server may need to operate in simplex and duplex modes simultaneously. To support multiple modes simultaneously, the kernel module keeps track of the system mode at a per connection granularity. At user-level, the server module uses our system call to check the mode of a connection. This check, i.e. system call, is only necessary while the transition from simplex to duplex is not fully complete — i.e., there are still existing simplex connections that have not yet terminated.

A conceivable improvement to our scheme would be to transition existing simplex connections to duplex mode as well, thus allowing for tolerance of multiple faults on the same connection. The simplex node's existing connection state and generated HTTP replies would have to be transferred to the new node,

which can be expensive. We extended our implementation to minimize the need for such a scheme by using the protocol semantics of HTTP/1.1 to ensure that after transition to duplex mode, existing simplex connections will be short-lived. HTTP/1.1 allows multiple requests to be pipelined on the same connection. However, the server has the option of terminating the connection at any point [Fiel99]. The server may close the connection cleanly using the HTTP “Connection: close” directive or by closing the connection at the transport level. The client should retry the pipelined requests for which it does not receive a reply [Fiel99]. We use this property to our advantage and force leftover simplex connections to be short-lived. After a server integration, the first request on each existing simplex connection is processed normally in simplex mode. However, when sending the reply, we notify the client at the HTTP level that the connection should be closed. As a result, the client will start a new connection for the processing of the rest of pipelined requests. Since these will be new connections, they will be automatically processed in duplex mode by our system. Hence, after the start of transition to duplex mode, our implementation will only process the single current active HTTP request of a connection in simplex mode. The rest of the requests will be processed in duplex mode, on a new connection.

4.5. Optimizations

Our performance evaluation of CoRAL (see Section 4.6) revealed two areas where CoRAL’s performance could be improved. 1) The backup server does not execute the application, and during fault-free operation its processing potential is largely wasted, especially for processor-intensive applications. 2) The reply

logging operations add unnecessary overhead for static, deterministically generated content. The application-level logging step can be safely eliminated for content that is generated deterministically. Based on the above findings, we implemented two performance optimizations for our scheme [Aghd03a]. The first, *dual-role servers*, improves throughput by distributing the primary and backup tasks among all the server hosts. The second, reduces the average overhead per request by allowing a more efficient scheme to be used with requests for static content.

4.5.1. Dual-Role Server Hosts

With our primary/backup scheme, the primary is likely to require significantly more processing than the logging that is performed on the backup, especially when serving dynamic content that requires significant processing in order to generate each reply. This type of content is typical for transaction processing e-commerce applications, where fault-tolerance is critical. For the experimental setup described in Section 4.6, Figure 4.6 shows the CPU cycles used per reply, where replies are generated using the WebStone [Mind02, Tren95] CGI benchmark. The large difference between the processing cycles used at the primary and backup servers indicate that the backup host is mostly idle, with its processing potential largely wasted. A simple solution to this problem is to distribute the primary server tasks and backup server tasks among all the hosts. Hence, each server host will serve as the primary for some requests and the backup for others [Time02, Borg83]. We refer to this scheme as *dual-role servers*. The distribution of requests between the two servers can be done using standard load balancing techniques, such as Round Robin DNS [Bris95], centralized load balancers [Cisc99a, Cisc99b], or redirection

schemes [Sury00].

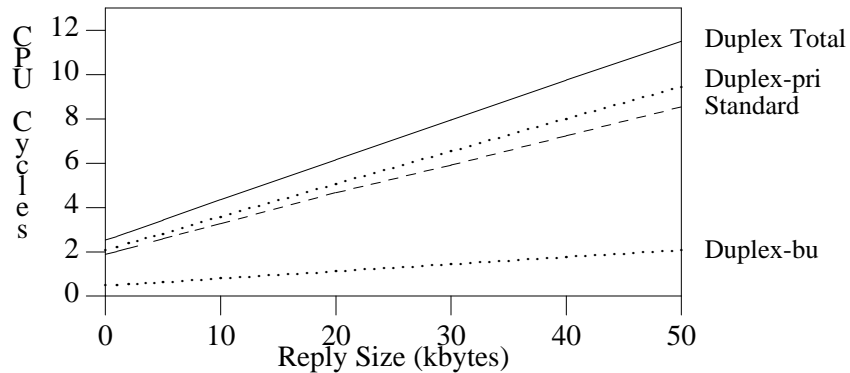


Figure 4.6: Server hosts CPU cycles (in million) per request for processing requests requiring dynamically-generated replies of different sizes. The primary and backup nodes of the system in duplex mode are depicted by *Duplex-pri* and *Duplex-bu* respectively. The *Duplex Total* line is the sum of the cycles used by the primary and backup per request.

CoRAL implementation without the dual-role optimization uses kernel modules and Apache server modules that are statically initialized to perform either the primary or backup functions [Aghd02]. For the dual-role server optimization, the kernel module must perform both functions simultaneously. Hence, for each packet received, the kernel module must dynamically determine whether it should process the packet as the primary or the backup. Similarly, it is also convenient for the web server processes (server modules) to dynamically determine whether to process requests at the primary or backup. Our solution is to use two separate TCP ports: one for incoming packets from clients and another for the forwarded packets from the other server. When a kernel module receives a packet on the public (client) port, it functions as the backup and forwards a copy of the packet to the

internal (forwarding) port of the other server. As a result, the forwarded packets arrive on a different TCP port number than the client packets, and the acting mode of the kernel module can be determined based on each packet's destination TCP port number. The same basic mechanism also works for the server module. Although the user-level server modules do not have access to the packet headers, they can determine the message destination address and their appropriate mode based on the socket where the request is received.

4.5.2. Efficient Handling of Static, Deterministic Content

As discussed previously, CoRAL is designed to handle non-deterministic dynamic reply generation. The reply is generated by the primary and then logged on the backup before its transmission to the client can begin. The logging is done over a dedicated TCP connection between the primary and backup. The primary waits for an explicit user-level acknowledgment from the backup before it begins to transmit the reply to the client [Aghd01, Aghd02]. Compared with transmission of the reply as soon as it is generated, our scheme results in increased latency. Specifically, most of the latency overhead of our scheme is due to the logging of the replies and increases with message size (see Section 4.6).

Much of the latency overhead of our scheme can be eliminated if the replies are deterministic — for example, if the replies are based on the contents of static data (files) available to all hosts. In that case, instead of logging replies, active replication [Schn90, Shen00] can be used, where both the primary and backup independently generate each reply.

A possible disadvantage of using active replication is increased CPU load on

the backup for generating the replies. However, deterministic replies of web services are often generated from static files and their generation is not processor intensive. Even processor intensive deterministic server applications (e.g. deterministic CGI scripts), typically have their results pre-computed and preserved in “cache” files or memory for performance reasons. With reply logging, our measurements have shown that the amount of processing required at the primary and backup hosts are similar when replies are generated from cached files [Aghd02]. Hence, the number of CPU cycles required by the backup server for logging the replies is approximately the same as the number of CPU cycles required by the primary to generate the replies. Thus, when processing associated with reply logging is eliminated, system performance improves (see Section 4.6.2.3.4).

The lack of synchronization between the primary and backup servers with this optimization can cause a degradation of performance for some requests. Since replies are no longer logged and reply messages are not exchanged between servers, the backup may fall behind the primary. The primary server may process a request, produce the reply, and send the reply to the client all *before* the backup server processes the same request. In such a case, the backup will receive client TCP acknowledgment packets *before* it has produced the corresponding TCP data packets. To maintain correctness for fault-tolerance, the backup kernel module drops these acknowledgments, allowing the primary and backup TCP states to converge before the acknowledgment packets are processed. This approach can lead to retransmission of some packets and an increase in the observed request processing time by some of the clients. This problem can be fairly common under

heavy load because both servers perform identical operations except that the backup performs the extra step of forwarding every client packet to the primary, making the backup the processing bottleneck of the system.

We have implemented an alternative optimization, henceforth referred to as *sync static* (synchronized static), that reduces the probability of retransmissions at a cost of some latency and processing overhead. With the sync static approach, the backup server sends a message containing the connection identifier to the primary upon the generation of each reply. The primary sends the reply to the client only after it receives the synchronization message from the backup. This prevents the primary from getting too far ahead of the backup.

The optimized version and the original reply logging version of our scheme can be used simultaneously on the same servers. The Apache web server provides a mechanism similar to a content-based (layer-7) router, where decisions about the processing of a request can be made based on the request URL in the HTTP header. Instead of routing requests to different servers as done by routers, Apache decides whether or not to use each module based on the request URL [Ste99]. If the request URL includes a path that has been designated to be non-deterministic content, our module that implements the reply logging is used. Otherwise, the module and the reply logging step is skipped. For example, servers can be setup where the URLs for non-deterministic content begin with *http://hostname/non-deterministic/*, and those for deterministic or static content begin with *http://hostname/static/*.

As mentioned before, the static optimization is best suited for cases where reply generation is not processor intensive. A conceivable further optimization for

processor intensive cases is to have the backup server log the requests and generate the reply *only* if the primary fails. As a result, processing cycles on the backup would not have to be used for reply generation during normal operation. Our original non-deterministic scheme also accomplishes this goal with the additional cost of sending the replies from the primary to the backup. Hence, the benefits of this possible approach are limited to only the cases where the replies are deterministic, processor intensive, and very large.

4.6. Performance Evaluation

In order to better understand the implications of design tradeoffs and implementation choices of CoRAL, we evaluated the performance of our scheme considering the overhead during normal operation as well as duration of any impact on the service when a fault occurs. The overhead during normal operation can be expressed in terms of increased response time, an increase in the number of server CPU cycles per request, and a decrease in the maximum request throughput that can be handled by a fixed number of servers. The response time (latency) will increase due mainly to the latency of forwarding the request by the backup and then logging the reply by the primary to the backup before sending the reply to the client. Some additional, relatively minor, factors that increase response time include address translation, sequence number mapping, and checksum recomputation. All the extra operations that increase the response time also account for the extra processing cycle per request. However, the extra latency is not directly proportional to the overhead processing cycles since part of the latency is due to communication between the server replicas and not simply extra

processing.

To a first order approximation, the response time of our scheme can be expressed by a very simple equation:

$$Response_Time = \alpha \times Reply_Length + \beta + Reply_Generation \quad (\text{Eqn } 1)$$

This equation also holds for the standard service implementation without any of our changes. *Reply_Generation* is the time to generate a reply on a server replica once the request has been received and thus this factor is not impacted by our scheme. Our scheme does impact α and β . The fixed time to set up the connection is β , while the latency per reply byte is α . While *Reply_Size* may impact *Reply_Generation*, neither of these factors is impacted by our scheme.

The total number of server processing cycles per request can be expressed with an equation that has the same form as Equation 1. The maximum throughput, when not limited by network bandwidth, is approximately inversely proportional to the processing cycles per request.

The rest of this Chapter presents the results of experimental evaluation of our implementations of CoRAL.

4.6.1. Experiment Setup

Unless otherwise noted, our measurements were performed on 350 MHz Intel Pentium II PCs interconnected by a 100 Mb/sec switched network based on a Cisco 6509 switch. The servers were running our modified Linux 2.4 kernel and the Apache 1.3.23 web server with logging turned on. For some experiments, the server replies were generated dynamically, using WebStone 2.0 benchmark's CGI

workload generator [Mind02, Tren95]. This benchmark randomly generates each reply byte. With this reply generator, where γ is the cost per byte of generating the reply, Equation 1 becomes:

$$Response_Time = \alpha \times Reply_Length + \beta + \gamma \times Reply_Length \quad (\text{Eqn } 2)$$

For other experiments, the server replies were static, read from a file whose contents did not change. In those cases, since the file was read repeatedly, it was cached in memory by the OS. Hence, the reply was effectively simply read from memory. With respect to Equation 2, this corresponds to $\gamma = 0$.

We used custom clients similar to those of the Wisconsin Proxy Benchmark [Alme98] for our measurements. The clients continuously generate one outstanding HTTP request at a time with no think time. For each experiment, the requests were for replies of a specific size as presented in our results. Internet traffic studies [Bres99, Cunh95] indicate that the median size for Web replies is typically around 15k bytes.

Measurements were conducted on at least three system configurations: *standard*, *simplex*, and *duplex*. *Standard* is the off-the-shelf system with no kernel or Web server modifications. The *simplex* system includes the kernel and server modifications but there is only one server, i.e., incoming packets are not multicast and outgoing messages are not logged to a backup before transmission to the client. The extra overhead of *simplex* relative to *standard* is due mainly to the packet header manipulations and bookkeeping in the kernel module. The *duplex* system is the full implementation of the scheme.

4.6.2. Failure-Free Performance

CoRAL introduces several new communication and processing steps that are performed in addition to the standard communication and processing that occur with a standard off-the-shelf web service. Incoming client TCP packets are forwarded by the Backup to the Primary, adding an extra hop to the path traveled by each incoming packet. The TCP/IP processing inside the kernel is actively replicated on both the Primary and Backup replicas. The transmission of replies to the client does not occur until a copy is logged by the Primary to the Backup, and an acknowledgement of the logging is received by the Primary. Each of these step adds to the overhead incurred by our system.

We performed several experiments to better understand the overheads of our system in failure-free operation. The key performance measures to consider include latency — the client observed time for receiving a reply for a request — and throughput — the amount of client requests that can be processed per unit time. Latency is important because clients often have an expectation of how long request processing should take. Hence, long latencies (longer than what the client expected) can lead to unsatisfied clients, and should be avoided. For web services the expectation for response time (latency) is typically in the sub-second to at most a few seconds range [Bail01]. Throughput is important because typically it is directly correlated with the service provider's income. A service provider's goal is to achieve the maximum possible throughput with its resources. A related metric is the amount of processing (CPU) cycles required to service each request. Assuming the service application is CPU bound (see discussion below), an increase in the processing requirement will result in either a lower system throughput or a higher

cost to the service provider (in terms of new/faster CPUs) to achieve the same throughput.

The system latency, throughput, and processing requirements are directly affected by the CoRAL overheads. The forwarding of incoming client TCP packets by the Backup to the Primary adds an extra hop to the path of each packet, increasing the latency of each request processed. The forwarding of packets also adds processing overhead since each received packet is copied and modified for sending. The active replication of TCP processing on both the Primary and Backup replicas also introduces processing overhead to the system. The reply logging step which allows for the handling of non-deterministic generated replies also adds overhead in terms of both latency and processing. The latency is also increased since the transmission of each reply to the client is delayed until the Primary sends a copy of the reply to the Backup and a user-level acknowledgement is received. This logging communication and the logging implementation described in Section 4.4.2.2 also requires some processing.

4.6.2.1. Application Reply Types and System Behavior

Web service application replies can be categorized into two types: static and dynamic. Static replies are generated deterministically, typically from contents of a file. The cost of generating these replies in terms of CPU cycles is relatively low. The files used for reply generation tend to be cached in memory, thus avoiding a disk access. As a result the latency for such replies is also typically relatively low and the throughput is often limited by the available bandwidth. Dynamically generated replies are typically non-deterministic and relatively processor intensive.

Therefore, the throughput is often limited by the amount of available CPU processing power.

With CoRAL, the content of replies is generated only at the primary (except with the static optimization discussed in Subsection 4.5.2). Hence, CoRAL's overheads are related to the size of the replies and *not* to the amount of processing required to generate the replies. There is a fixed processing overhead that is incurred regardless of the reply type. As a result, the overhead of CoRAL *relative* to a standard server is larger for static replies (which require few cycles to generate) than the more processor-intensive dynamic replies. In Section 4.5.2 we showed how to optimize CoRAL and reduce the overhead for static content. However, in general, practical applications that require a high level of reliability and fault-tolerance (e.g., transaction-based systems) typically involve dynamically generated replies. Hence, our experiments with dynamically generated replies are more representative of a practical setting.

4.6.2.2. Dynamically Generated Replies

As mentioned earlier, practical systems that require a high level of reliability and fault-tolerance are typically processor intensive. Static content results presented in Section 4.6.2.3 show the baseline case where the generation of replies requires minimal processing. In this section we present results for an example application where more CPU cycles are spent on generating a reply than the static case — a CGI script that dynamically generates the replies. As described in Section 4.6.1, for these experiments our clients were configured to send requests for the CGI content. The web server used the CGI script that is part of WebStone

2.0 benchmark [Mind02, Tren95]. This CGI workload generator generates each reply byte using a random number generator. In addition, as with typical CGI processing, a new server process is created (forked) to handle the request. Hence, the server CPU utilization is significantly higher than the CPU utilization for generating static content. This behavior is more indicative of targeted practical applications such as transaction processing where increased system reliability and fault-tolerance is required. We measured the client observed latency for individual requests, peak system throughput in terms of requests per second and Mbytes per second, and processing overhead incurred due to our fault-tolerance implementation.

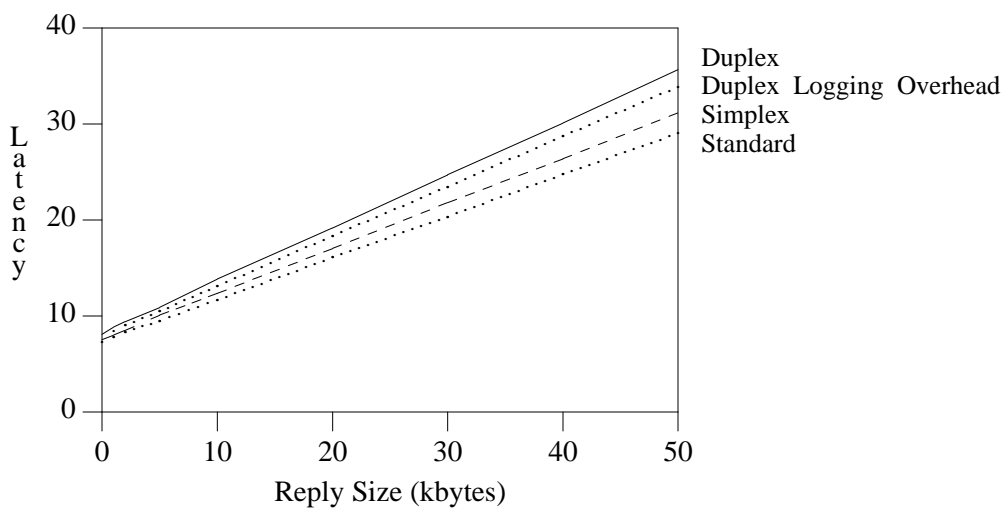


Figure 4.7: Average latency (ms) observed by a client for different reply sizes and system modes.

4.6.2.2.1. Latency Overhead with Dynamic Replies

Figure 4.7 shows the average latency on an unloaded server and network from the transmission of a request by the client to the receipt of the corresponding reply. There is only a single client on the network, with a maximum of one outstanding request. These results match Equation 2, with $\gamma = 0.34 \mu\text{s/B}$, and for (standard, simplex, duplex), respectively, $\alpha = (0.10, 0.14, 0.21) \mu\text{s/B}$, $\beta = (5.59, 5.86, 6.40) \text{ms}$.

The higher value of the factor α for *duplex* compared to *standard* means that the absolute latency overhead increases with increasing reply size. The extra processing per reply byte is due mostly to the logging of the reply. In Figure 4.7, the difference between the ‘‘Duplex Logging Overhead’’ line and the ‘‘Standard’’ line is the time to transmit the reply from the primary to the backup and receive an acknowledgement at the primary. As the figure shows, this time accounts for most of the duplex overhead. In practice, taking into account server processing and Internet communication delays [Matr00], server response times of tens or even hundreds of milliseconds are common. Hence, the increased latency of a few milliseconds (6.6ms for 50KB replies) will have negligible impact on the service observed by the Internet clients.

4.6.2.2.2. Processing Overhead with Dynamic Replies

We measured the CPU cycles used by a server host in several different settings in order to identify CoRAL’s processing overhead. The measurements were performed using a driver [Pett03] which utilizes the processor’s performance monitoring counter registers. Specifically, we measured

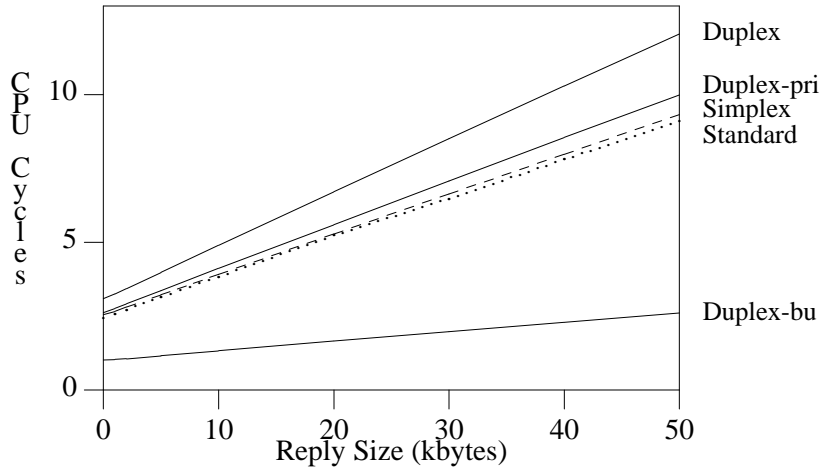


Figure 4.8: Used CPU cycles (in million) by different system modes for processing requests for different reply sizes. The primary and backup nodes of the system in duplex mode are depicted by *Duplex-pri* and *Duplex-bu* respectively. The Duplex line is the summation of primary and backup results.

global_power_events [Inte04] which accumulates the time during which the processor is not stopped. First we measured the number of cycles used by an idle host (just running the OS and minimal system processes) in a given amount of time (e.g. 10 seconds). We then measured the number of cycles used by a host while processing a known number of requests (e.g. 1000) in the same time period. The actual number of cycles used by our scheme was obtained by deducting the cycles used by an idle host from the cycles used by the host processing requests.

Figure 4.8 shows the CPU cycles used by the servers to receive one request and generate a reply of a given size. As mentioned earlier, these results can be modeled by an equation that has the same form as Equation 2:

$$Cycles_Used = \omega \times Reply_Length + \psi + \phi \times Reply_Length \quad (\text{Eqn } 3)$$

The measured results fit the values: $\phi = 120$ cycles/B, and for (standard, simplex,

duplex), respectively, $\omega = (10, 12, 56)$ cycles/B, $\psi = (1.89, 2.0, 2.55)$ Mcycles.

Comparing *duplex* to *standard*, the component of the overhead that depends on the reply size (the change in ω) is 46 cycles per reply byte. This overhead consists of 14 cycles/B on the primary host and 32 cycles/B on the backup. This overhead component is larger on the backup because the extra work on the primary per reply byte is simply to transmit (log) it to the backup. On the other hand, on the backup the extra work is to receive the reply byte, transmit it (with the kernel module dropping it before it is physically transmitted to the network), and handle as well as forward the client acknowledgements for the reply.

Based on Figure 4.8, the relative overhead of *duplex* versus *standard* is in the range of 32%-35% for the entire range of reply sizes. This relative overhead is heavily dependent on the cost of generating the reply (the parameter ϕ in Equation 3. For example, if the reply is static, generated from files that are cached in memory, the value of ϕ is essentially 0. In this case, the measured results (Subsection 4.6.2.3.1) fit the values: for (standard, simplex, duplex), respectively, $\omega = (10, 13, 56)$ cycles/B, $\psi = (0.385, 0.423, 0.949)$ Mcycles. While the absolute overhead for duplex is essentially the same as with dynamic replies, the relative overhead reaches about 310% for 50KB replies. For most applications where reliability is important (e.g., banking), there is likely to be significant processing required to generate the reply. Hence, with respect to relative processing overhead, the results above for *dynamic* content are more representative of practical deployment. As shown in Section 4.6.2.3.4, lower relative overheads can be achieved using the optimization for static content discussed in Section 4.5.2.

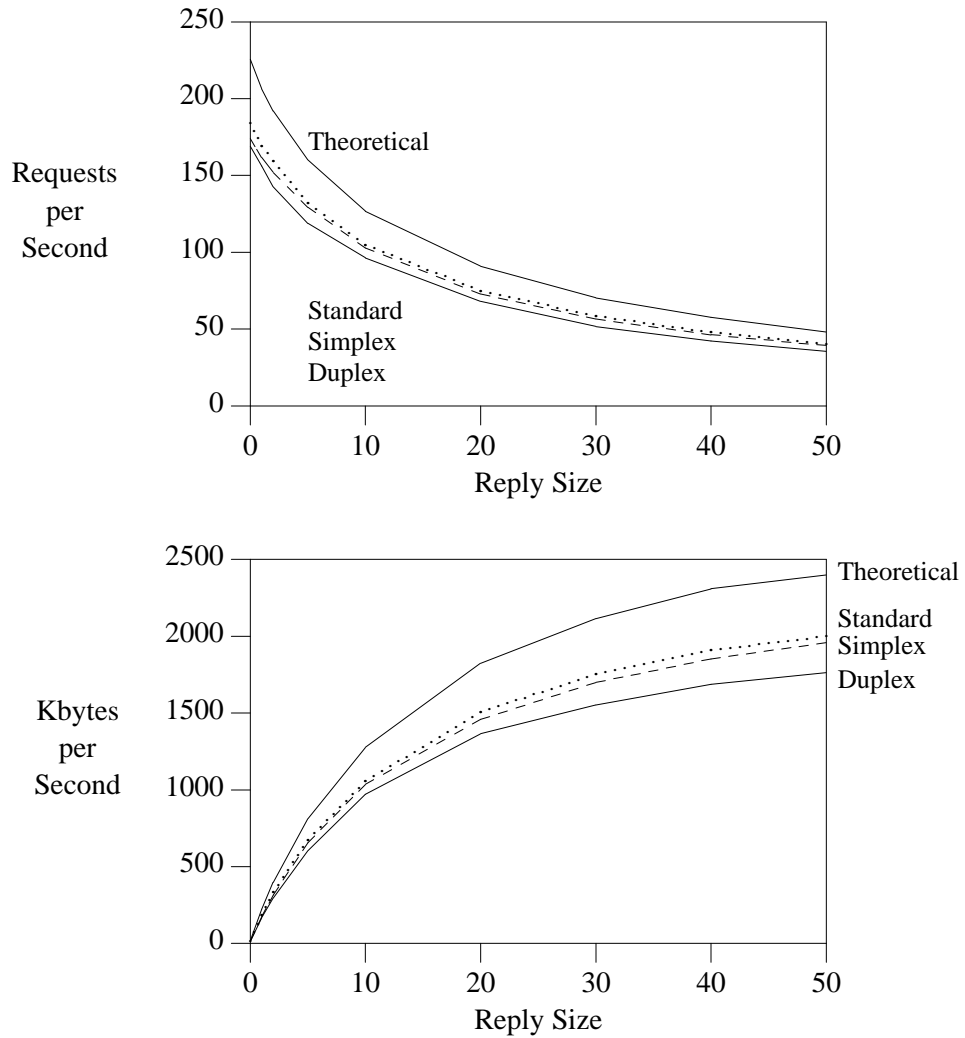


Figure 4.9: Peak system throughput (in requests and kbytes per second) for different message sizes (kbytes) and system modes. “Theoretical” line represents the theoretical throughput of a standard server which uses an equivalent number of processing cycles as the duplex system.

4.6.2.2.3. Throughput with Dynamic Replies

With the experimental setup for dynamically generate replies, the maximum throughput is limited by the number of CPU cycles required per request. Figure 4.9 presents the measured peak throughput for different system

configurations. As shown in Figure 4.8, with the *duplex* configuration, the number of CPU cycles per request on the primary is just slightly higher than with the *standard* configuration (less than 11% higher). Hence, the maximum throughput with *duplex* is also within 11% of the maximum throughput of *standard*. However, this comparison is not “fair” since *duplex* uses two servers while *standard* uses only one. With the same amount of resources (two server hosts), a *standard* system can achieve twice the throughput of a single server. Specifically, the difference between the $2 \times \textit{standard}$ line and the *duplex* line represents the real throughput overhead of our scheme — *duplex* achieves only 44%-46% of the maximum throughput of *standard* on two hosts.

The *theoretical* line represents the theoretical throughput of a standard server where in addition to a single server, the cycles used by the backup server in our scheme is also used for generating replies in standard mode. The values were calculated using our processing cycle measurements for our backup server and the cycle measurements of a standard unreplicated server.

As discussed in Section 4.5.1, the large throughput overhead of the *duplex* configuration is due to the fact that only the primary processes the requests and the backup is mostly idle. Section 4.6.2.2.4 presents an evaluation of the *dual-role* optimization (Section 4.5.1) that keeps both server replicas busy.

Since the primary server sends the replies to the backup and to the clients on the same physical link, it is possible for this link to become a bottleneck. This phenomenon is unlikely to occur for processor intensive applications (i.e. dynamic replies) but it can occur for network bound applications such as cached static file replies. In such cases, the throughput of our system can be improved by using a

secondary network interface on each server for the reply logging (see Subsection 4.6.2.3.2).

4.6.2.2.4. Evaluation of the Dual-Role Servers Optimization

In Section 4.5.1 we described our dual-role optimization where each server host can simultaneously act as both the primary and backup for different requests. As a result, idle cycles on the backup can be used to process requests and generate replies (as the primary), increasing the overall throughput of the system. Figure 4.10 presents the throughputs of a standard server and CoRAL with and without the dual-role optimization. For these measurements clients were manually configured so that half of the requests were sent to each server. In practice, a load-balancing mechanism should be used to distribute the requests amongst the servers.

The *dual role* results show significant throughput improvement over duplex results (Figure 4.10) — 70%-72% of the *Standard* throughput (on two hosts) is achieved. As mentioned earlier, the performance improvement is due to the use of otherwise idle backup cycles for processing of requests. The difference between the $2*standard$ line and the *duplex* line represents the throughput overhead of our original scheme. The *dual role theoretical* line shows the theoretical upper bound of the dual-role scheme. The values were calculated using measurements of required CPU cycles for the processing of a single request and the known processing speed of our servers. The small difference between the theoretical (calculated) and experimental values are likely due to the increased number of context switches that occur during high loads.

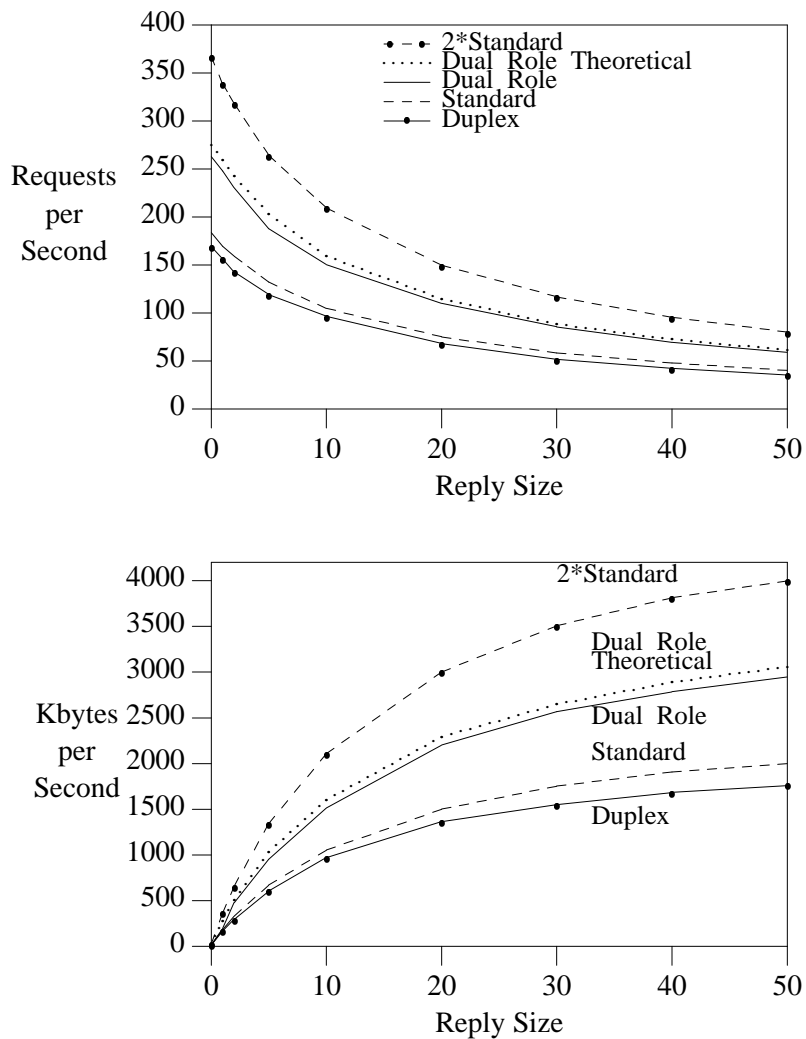


Figure 4.10: Peak system throughput (in requests and kbytes per second) for different reply message sizes (kbytes). Replies generated with WebStone 2.0 CGI benchmark.

4.6.2.3. Replies Based on Static Content

For static content experiments, the web server generated the replies using contents of files available on the local host. Clients made requests to files of specific sizes as noted in each experiment. Since repeated requests were made for the same files, file contents were cached in memory and disk I/O operations were

not involved. We measured the client observed latency for individual requests, peak system throughput in terms of requests per second and Mbytes per second, and processing overhead incurred due to our fault-tolerance implementation. The setup for the experiments were described in Section 4.6.1. Results are presented in the rest of this subsection.

4.6.2.3.1. Latency Overhead for Static Content

For latency measurements, a single client sent a single outstanding request to an unloaded server and network, and the time from the transmission of the request to the the receipt of the full corresponding reply was measured on the client. This experiment was repeated one thousand times for each reply size, and the averages were calculated.

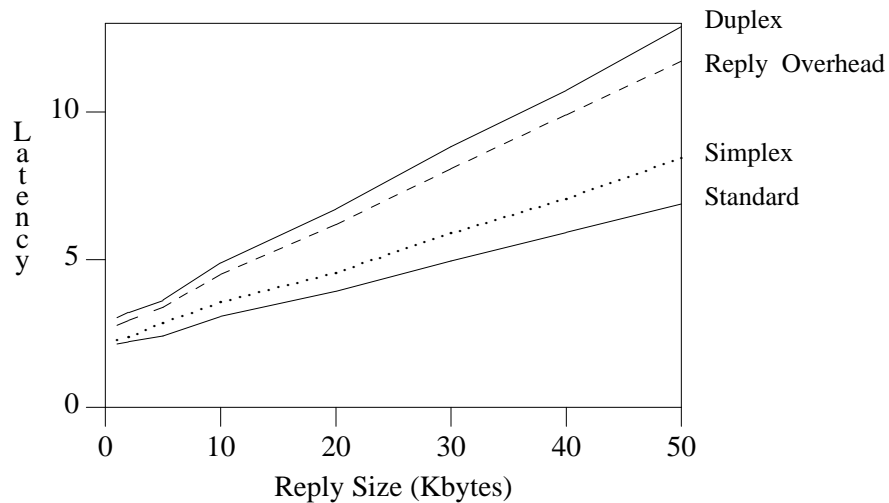


Figure 4.11: Average latency (ms) observed by a client for different reply sizes and system modes. The *Reply Overhead* line depicts the latency caused by replication of the reply in duplex mode.

Figure 4.11 shows the average client observed latency. The results show that

the latency overhead relative to the standard system increases with increasing reply size. This is due to processing of more reply packets and the associated overhead of modifications made to each packet header. The difference between the *Reply Overhead* line and the *standard* line is the time to transmit the reply from the primary to the backup and receive an acknowledgement at the primary. This time accounts for most of the duplex overhead. Note that these measurements exaggerate the relative overhead that would impact a real system since: 1) the client is on the same local network as the server, and 2) the requests are for (cached) static files. In practice, taking into account server processing and Internet communication delays [Matr00], server response times in the order of a hundred milliseconds or greater are common. The absolute overhead time introduced by our scheme remains the same regardless of server response times and therefore our implementation overhead is only a small fraction of the overall response time seen by clients.

4.6.2.3.2. Throughput with Static Content

We conducted experiments to measure the peak throughput achieved by the system. Several experiments were performed, each for a fixed reply size. For these measurements, multiple clients simultaneously send requests to the server. There is one outstanding request per client. Upon the receipt of a reply, the client continues and transmits another request. The total number of client requests processed per unit time by the server were measured.

The number of clients were fixed for each experiment, and ranged from 8 for large (50k) replies to 32 for small (1k) replies. The number of clients for each

experiment was carefully chosen. If the number of clients for an experiment is too small, the server will receive fewer requests than it is capable of handling and the throughput results will be artificially low. If the number of clients is too large, the server will receive more requests than it can handle, causing packets to be dropped. This will cause TCP to backoff and the client to send requests at a lower rate. Since the number of experimental clients (tens) are much smaller than in practice (hundreds or thousands), and new experiment requests are sent by a client only after the previous request has been serviced, a small backoff by the experimental clients will cause an exaggerated drop in the number of transmitted requests and thus system throughput. Hence, we manually varied the number of clients to find the best (highest throughput) setting for each (reply size) experiment.

Figure 4.12 shows the peak throughput of a single pair of server hosts for different reply sizes. The throughputs of *standard* and *simplex* (in Mbytes/sec) increase until the network becomes the bottleneck. However, the duplex mode throughput peaks at less than half of that amount. This is due to the fact that on the primary, the sending of the reply to the backup by the server module and the sending of reply to the clients (Figure 4.3) occur over the same physical link. Hence, the throughput to the clients is reduced by half in duplex mode. To avoid this bottleneck, the transmission of the replies from the primary to the backup can be performed on a separate dedicated link. A high-speed Myrinet [Bode95] LAN was available to us and was used for this purpose in measurements denoted by “duplex-mi”. These measurements show a significant throughput improvement over the duplex results, as a throughput of about twice that of duplex mode with a single network interface is achieved.

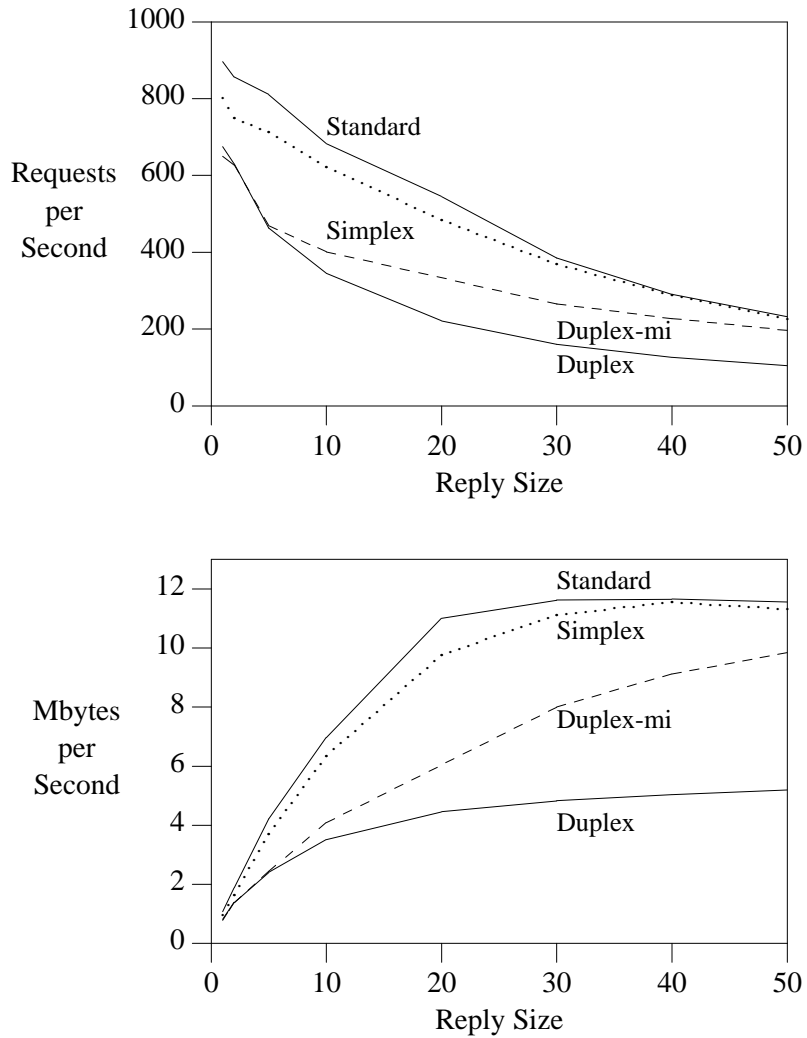


Figure 4.12: System throughput (in requests and Mbytes per second) for different message sizes (kbytes) and system modes. Duplex-mi line denotes setting with multiple network interfaces for each server - one interface is used only for reply replication.

System Mode	1kbyte reply				10kbyte reply				50kbyte reply			
	user	kernel	total	cpu%	user	kernel	total	cpu%	user	kernel	total	cpu%
Duplex (primary)	190	337	527	100	193	587	780	77	224	1548	1772	53
Duplex (backup)	147	330	477	91	158	615	773	76	185	1790	1958	58
Duplex-mi (primary)	192	353	545	100	198	544	742	85	225	1283	1508	85
Duplex-mi (backup)	147	355	502	93	152	545	697	80	169	1124	1293	72
Simplex	186	250	436	100	191	365	556	99	208	871	1079	70
Standard	165	230	395	100	166	342	508	99	178	730	908	60

TABLE 4.3: Breakdown of used CPU cycles (in thousands) - cpu% column indicates CPU utilization during peak throughput.

4.6.2.3.3. Processing Overhead with Static Content

CoRAL’s processing overhead was evaluated by taking advantage of the processor’s performance monitoring registers [Inte04], using the procedure described in Subsection 4.6.2.2.2. Table 4.3 shows the CPU cycles used by the servers to receive one request and generate a reply of different sizes. For each configuration the table presents the kernel-level, user-level, and total cycles used. The “cpu%” column shows the cpu utilization at peak throughput, and indicates that the system becomes CPU bound as the reply size decreases. This explains the throughput results, where lower throughputs (in Mbytes/sec) were reached with smaller replies.

Based on Table 4.3, the duplex server (primary and backup combined) can require more than four times (for the 50KB reply) as many cycles to handle a request compared with the standard server. However, as noted earlier, the likely applications of this technology is for dynamic content. With dynamic content,

replies are likely to be smaller and require significantly more processing. Hence, the actual relative processing overhead can be expected to be much lower than the factor of 4 shown in the table (see Section 4.6.2.2.2).

4.6.2.3.4. Optimization for Deterministic or Static Content

Section 4.5.2 presented an optimization for more efficient handling of deterministic or static content. Much of the overhead of our scheme is due to the logging of replies from the primary to the backup. Furthermore, when the reply is simply the contents of a file that is usually cached in memory, generating the reply requires few CPU cycles. Hence, instead of logging the reply, it is more efficient to generate the reply at both the primary and backup.

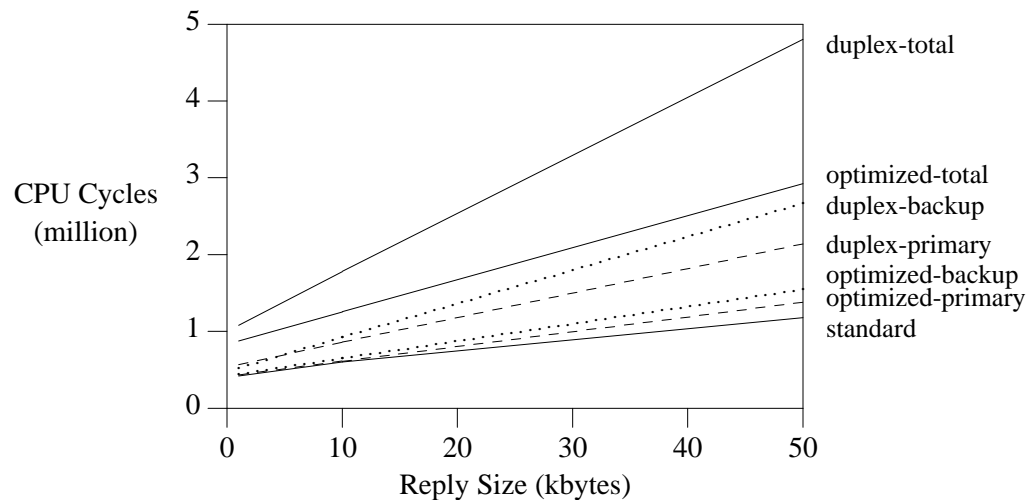


Figure 4.13: Processing cycles per request, with replies generated from cached static files. Two schemes are compared: the full duplex CoRAL and the version of CoRAL optimized for static content. In each case, results include the cycles at the primary, at the backup, and total cycles used.

For replies generated from cached static files, Figure 4.13 compares the CPU

cycles used per request with the full duplex CoRAL and with the version of CoRAL optimized for static content. These measurements show that with the optimization for static content, compared to the full duplex CoRAL, there is a 45% reduction in the cycle count per reply byte, and an 18% reduction in the fixed cost to set up the connection. The results for the full duplex CoRAL shown in Figure 4.13 are different from the results shown in Table 4.3, even though it is for the same basic configuration. The reason for this difference is that the measurements were conducted on different versions of the operating system with different device drivers. However, assuming that the relative reduction in cycle count for these two similar configurations would be approximately the same, it is possible to compute the cycle counts for the optimized CoRAL that corresponds to the result for the full CoRAL shown in Table 4.3. Applying the results of this computation to Equation 3, this yields parameter values of: $\phi = 0$, $\omega = 31$, $\psi = 0.778$. These parameters can be compared with the parameters discussed in Subsection 4.6.2.2.2. These results show that this optimization can significantly reduce the overhead of our scheme for static, deterministic content. In particular, with this optimization, for reply sizes of up to 50KB, the relative overhead of our scheme for static content is below 163%.

Figure 4.14 shows the average request latency as observed by a client. The *static* line shows the latency with the static optimization. There is very little latency overhead compared to the unoptimized *duplex* implementation because most of our latency overhead (which is due to reply logging — *reply overhead* line) is eliminated.

Figure 4.15 shows the distribution of request-reply latencies for the *standard*,

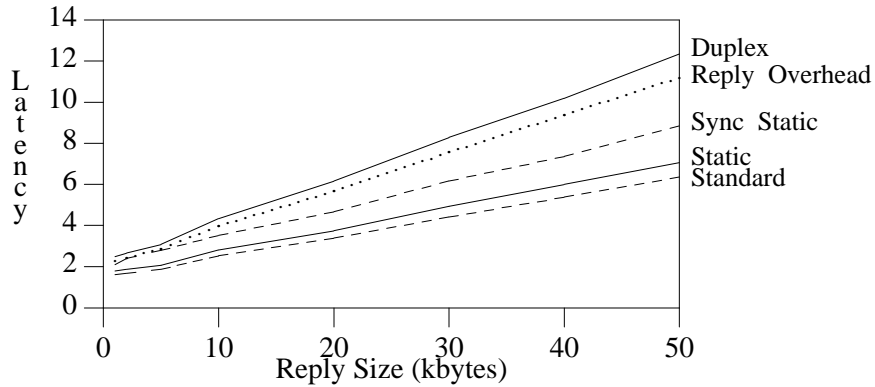


Figure 4.14: Average latency (ms) observed by a client for dynamically generated replies with different sizes. The *Reply Overhead* line shows the fraction of *Duplex* mode overhead due to reply replication. *Static* and *Sync Static* depict the latency with optimization.

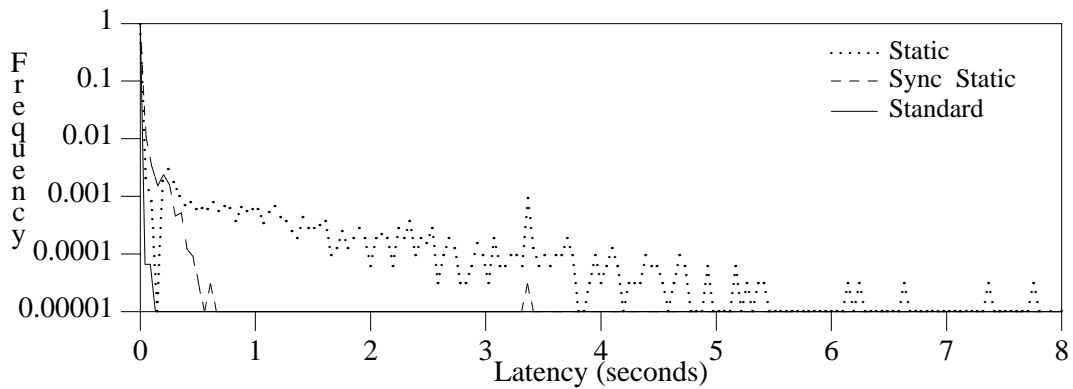


Figure 4.15: Latency distribution for 5 Kbyte static replies at 550 requests/second throughput. (Note the log scale on the Y axis).

static, and *sync static* schemes for 5Kbytes static replies and a throughput of 550 requests per second. While the overwhelming majority of requests are processed within a few milliseconds, with the *static* optimization, under heavy load, a tiny fraction of requests result in latencies on the order of few seconds. This is due to an increased probability of required retransmissions as discussed in

Subsection 4.5.2. The figure also shows that the *sync static* optimization reduces the probability of the long latencies. Figure 4.14 shows that the reduced latency variance of the *sync static* scheme compared to the *static* scheme comes at the cost of increased average latency.

Figure 4.16 shows the peak throughput of our system with replies being generated from cached static files. For large replies, the large throughput difference between a standard server and our original *duplex* scheme is largely due to the network becoming a throughput bottleneck due to the logging of replies. We showed in Subsection 4.6.2.3.2 that this situation can be greatly improved by using two network interfaces on each server, with one being dedicated for the reply logging. The *duplex-mi* line shows the benefit of using such a dedicated connection.

The *static* optimization eliminates reply logging and thus, as shown in Figure 4.16, exceeds the performance of the *duplex-mi* scheme, without using a dedicated network connection. The increased throughput compared to the *duplex-mi* scheme is due to the elimination of the processing overhead involved in logging the reply. The *sync static* line in Figure 4.16 shows that, due to the increased processing required, there is also a throughput cost for reducing the latency variance relative to the *static* scheme.

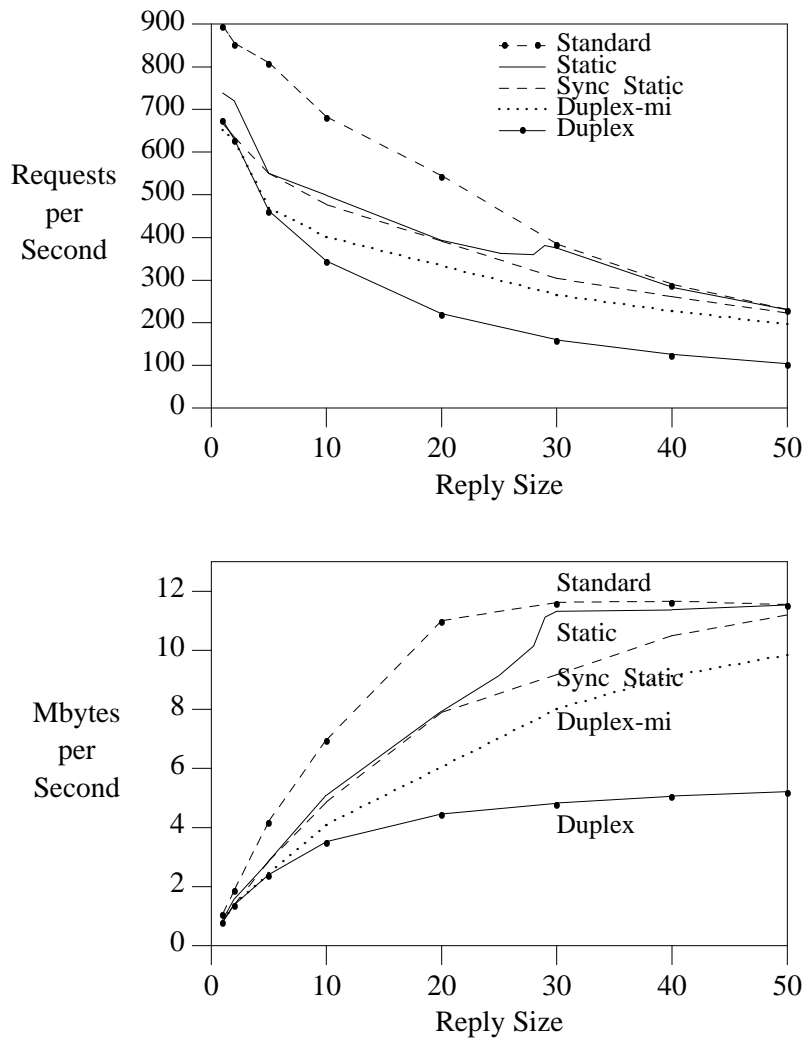


Figure 4.16: Peak system throughput (in requests and Mbytes per second) for different reply message sizes (kbytes). Replies generated from cached static files. *Duplex-mi* shows results for duplex system with multiple network interfaces. *Static* and *Sync Static* depict peak throughput for optimizations.

4.6.3. Service Failover and Recovery

When a server replica detects that the other member of the server pair has failed, it reconfigures itself to continue to operate in simplex mode (see

Section 4.4.4). These operations require time, and the system may be unavailable to clients while the failover and recovery operations take place.

To evaluate this system failover, fail-stop faults were emulated by physically disconnecting a server host from the network. For these experiments, the workload was generated by eight client processes, each continuously sending an HTTP request for a 1 Kbyte dynamically generated reply.

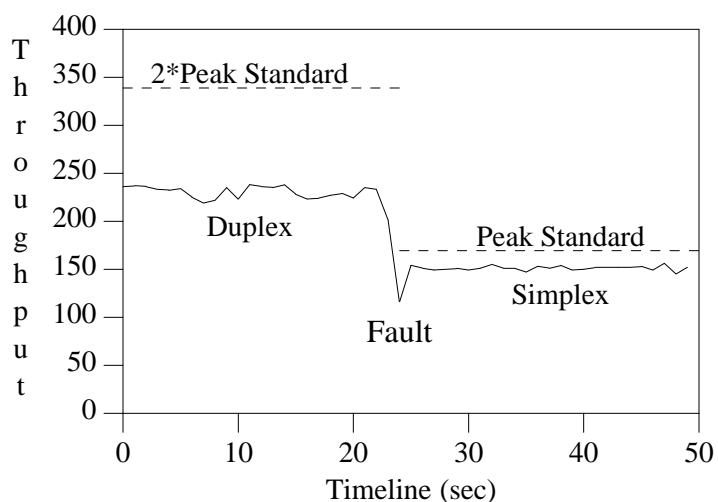


Figure 4.17: System throughput (in requests per second) for 1 kbyte HTTP replies before and after a fault. Dashed lines indicate the upper bound, i.e., the peak throughputs of one and two standard servers serving the same content.

Figure 4.17 shows the system throughput before and after a failover, measured at one second intervals. The system operates at the maximum duplex system throughput prior to the fault. Since in duplex mode replies are not actually generated on the Backup, two servers in dual-role duplex mode can process more requests per unit time than a single server in simplex mode. Thus, the throughput of the system decreases following reconfiguration to simplex mode.

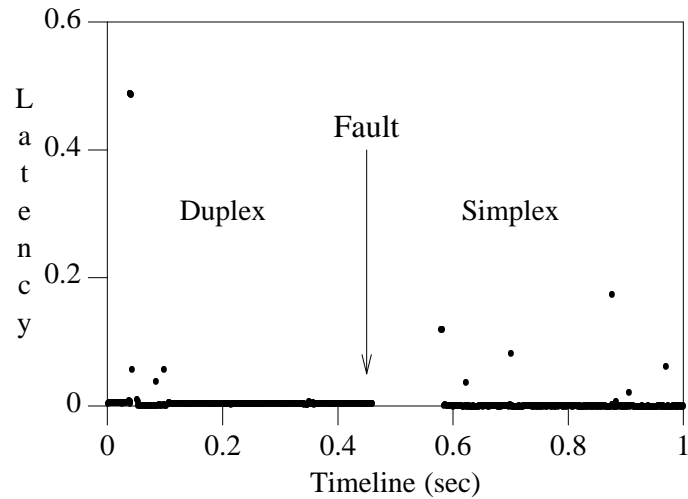


Figure 4.18: Client observed request latencies (in seconds) for 1 kbyte HTTP replies before and after a fault. During failover, the system is unavailable for a fraction of a second.

When a server host fails, there is a short period of time during which no new requests are processed — the system is unavailable. This is the time required for fault detection and failover from duplex to simplex mode. Figure 4.18 shows typical client observed request latencies before and after a fault. Although clients continuously transmit requests, there is a gap at the fault point where no requests are processed — the system is unavailable for a short period of time. Our measured failover transition time from fault detection in duplex mode to system execution in simplex mode is just 1.5 milliseconds (see Table 4.4). Since the transition time is short, most of the overall failover time is the time for fault detection. The heartbeat rate determines the fault detection time but generating and processing heartbeats uses CPU resources. Hence, the choice of this rate is a tradeoff between failover speed and overhead during normal operation. For our evaluation, we used a heartbeat rate of 100 milliseconds and a fault was assumed

when two consecutive heartbeats were missed. As a result, the unavailability period was between 100 and 200 milliseconds.

In the worst case, when a server replica fails, a few packets may be lost. The impact of this will be exactly the same as the impact of packets lost in the network — the client may experience an additional delay that is approximately equal to the TCP retransmission timeout. Such packet losses may occur during the brief (fraction of a second) “unavailable time” mentioned above. However, all requests are eventually received by a working server replica and all replies are eventually delivered to the client.

Operation	Latency (ms)
Kernel Notification and Operations	0.3
User-level Process Notification and Operations	0.5
Identity Preservation	
gratuitous ARP	0.5
outside host ping	0.2
Total	1.5

Table 4.4: Breakdown of failover latency measured from the time a fault is detected to start of system operation in simplex mode. This table does not include the time to regenerate replies that were not logged prior to the failure.

After a failover, it is desirable for the system to return to duplex operation, allowing for the handling of other faults that may occur. Our implementation includes a mechanism for integrating a new server with an active simplex server (Section 4.4.5). We measured the required time for each of the key operations involved. Table 4.5 shows a breakdown of latency for integration of a new server with a simplex server. Most of the overall time is due to forking of the processes that are used for communication between the server for reply logging. In our

implementation, the logging processes are forked and initialized only after an initial handshake between the simplex host and the new server. This allows for minimal resource requirements during simplex operation. The fork and initialization latency overhead can be reduced by eliminating the need to fork processes during server integration. This can be accomplished, at a cost of increased resource usage, by forking these processes in simplex mode before they are actually needed.

The other significant amount of time is due to exchange of configuration information, i.e., server TCP/IP addresses, over a TCP connection. This is necessary because we assumed that the active simplex server has no knowledge regarding the new server that is joining it. Static configuration of server pairs can reduce or eliminate this overhead. The identity restoration step consists of removing an aliased IP address from the simplex node, mapping this IP address to the new server, and informing the router using our reliable gratuitous ARP implementation described in Section 4.4.4.

Operation	Latency (ms)
Identity Restoration	0.3
Exchange Configuration Info Over TCP	3.9
Fork and Initialize Processes	6.8
Total	11.0

Table 4.5: Breakdown of latency for integration of a new server with a working simplex server node to recover back to a duplex system.

4.6.4. Implementation Comparison: User-Level versus Kernel Support

As mentioned earlier in Chapter 4, the first implementation of CoRAL was based on user-level proxies, without any kernel modifications. Two levels of proxies implemented the required functionality. The first layer (raw proxies) had packet header rewriting capabilities, and implemented the connection replication portion of our scheme. The second layer (tcp proxies) implemented the message logging requirements at the user-level. The performance analysis of our proxy based implementation (discussed below) showed that our scheme introduces a minimal latency overhead to standard off-the-shelf servers [Aghd01]. However, the processing cost of our user-level implementation led us to the more practical and efficient implementation using kernel-level support. In the rest of this section we present details of our performance analysis for our user-level implementation and compare the overheads of our user-level and kernel-level implementations.

The proxy-based experiments were performed on 350 MHz dual Intel Pentium II PC's running Solaris 7 and connected via a switched network using 100 Mbit/Sec Ethernet cards and a Nortel 350 Baystack 10/100MB switch. In all experiments the raw and TCP proxies of each cluster (primary/backup) ran on the same machine. We used custom clients and servers similar to those of Wisconsin Proxy Benchmark [Alme98] for our measurements. The client generates continuous HTTP requests and in response the server sends a reply of predetermined size without any processing.

Figure 4.19 compares the request latency times as measured by the client for 1 kbyte messages. We ran the same experiment with our system in duplex mode and simplex mode. We also ran the experiment with direct client and server

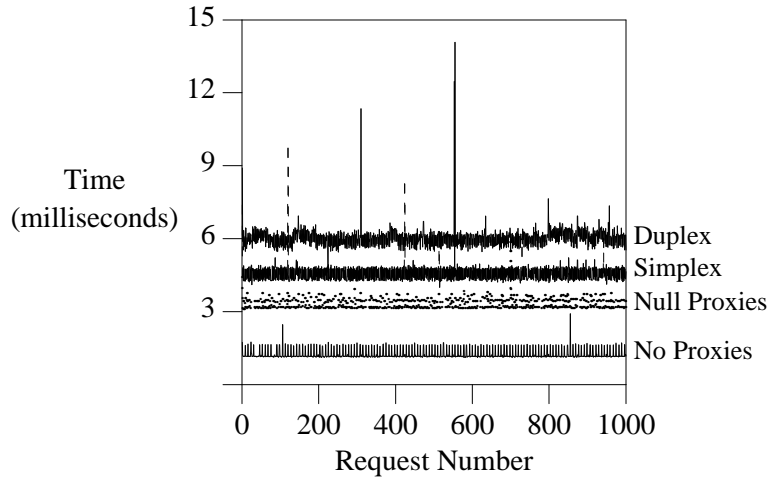


Figure 4.19: Comparison of client measured response times for different system modes

Mode	Average (ms)	Median (ms)
Duplex Mode	6.05	6.08
Simplex Mode	4.56	4.44
Null Proxies	3.27	3.19
No Proxies	1.23	1.16

Table 4.6: Average and median response times for different system modes

connections (“no proxies”) and on a simplex mode system with proxies that just relay the packets (“null proxies”). The performance of the system with null proxies was evaluated in order to evaluate the overhead introduced by our decision to use the two proxy approach. The difference between the simplex and duplex modes shows the replication overhead. We have verified that the “spikes” in Figure 4.19 are not caused by processing time of our code, message loss, or retransmission. We believe these spikes to be caused by elements outside of our implementation, such as the operating system. Table 4.6 shows the average and

mean request times for each experiment.

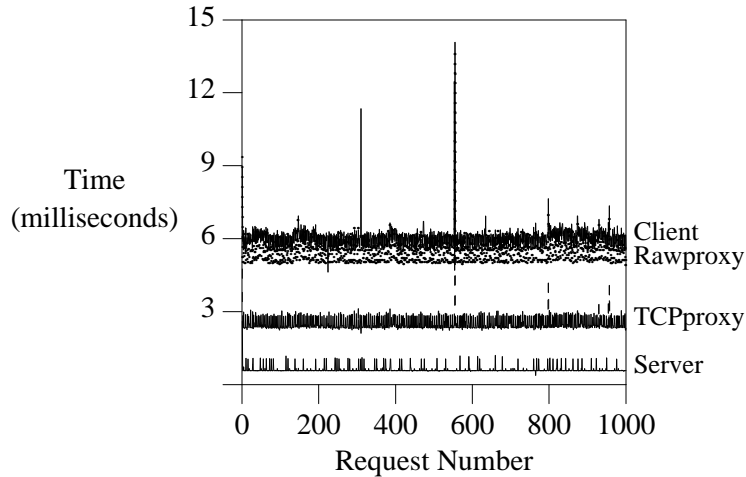


Figure 4.20: Component breakdown of duplex mode response time for 1 Kbyte replies

Component	Avg (ms)	Median (ms)
Client	6.05	6.08
Raw Proxy	5.42	5.26
TCP Proxy	2.52	2.40
Server	0.61	0.57

Table 4.7: Breakdown of average and median response times for 1 Kbyte replies in duplex mode

Figure 4.20 and table 4.7 show the breakdown of where the time is being spent for each message. They show the times for the raw proxy, TCP proxy, and the server. Note that the figure depicts an exaggerated view of overhead that is introduced by the proxies since we use a custom server that does not do any processing. In practice, taking into account server processing and Internet communication delays, server response times of hundreds of milliseconds are common. The absolute overhead time introduced by the proxies remains the same

regardless of the server response time. Therefore, in practice our implementation overhead is only a small percentage of the overall response time.

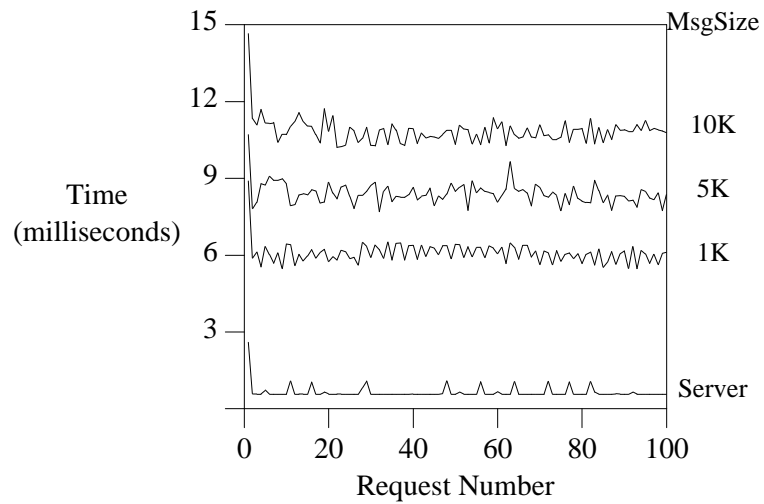


Figure 4.21: Duplex mode response times for different HTTP message sizes

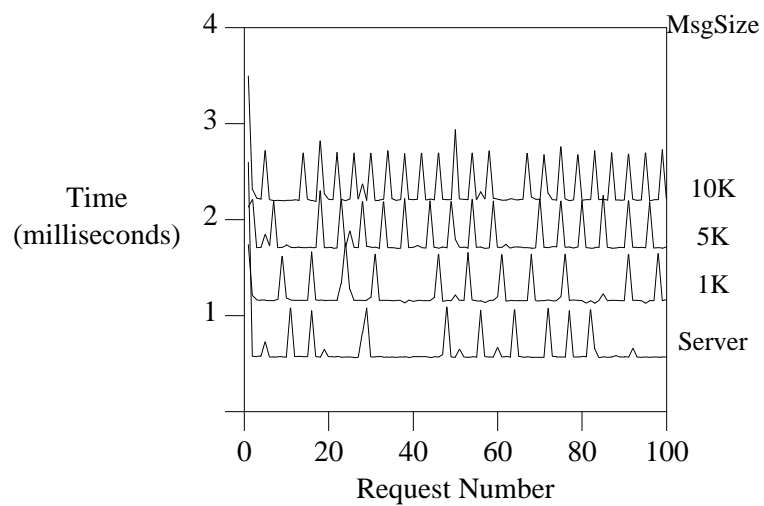


Figure 4.22: Standard server (without proxies) response times for different HTTP message sizes

The comparison of request times for different HTTP reply message sizes

Mode	Size (kbytes)	Avg (ms)	Median (ms)
Duplex	1	6.05	6.08
Duplex	5	8.41	8.36
Duplex	10	10.85	10.87
No Proxies	1	1.23	1.16
No Proxies	5	1.81	1.71
No Proxies	10	2.34	2.21

Table 4.8: Average and median request times for different HTTP message sizes

shows that our system scales in the same manner as a non replicated system. Figure 4.21 shows the client request times for our system in duplex mode and Figure 4.22 shows the times of the same requests for direct client to server communication. The server processing time is also shown in both figures. Table 4.8 lists the average and mean times. Our system introduces an increased connection setup overhead (difference between server time and small message size) and a slightly larger per packet processing overhead (difference between small and large messages). The figures show that the processing overhead increases linearly in respect to message size in both cases and therefore both systems scale in the same manner with respect to message size and number of TCP/IP packets.

Our preliminary performance evaluation of our proxy-based user-level implementation indicated insignificant latency overhead. The overall latency overhead (similar to those of our kernel-level implementation presented in Section 4.6.2.3.1) is on the order of few milliseconds — acceptable for most network service applications. However, further experiments showed that the user-level implementation incurs a large processing overhead and reduced throughput.

Implementation	Primary	Backup	Total
User-level Proxies	1860	1370	3230
Kernel/Server Modules	337	330	667

TABLE 4.9: User-level versus kernel support — CPU cycles (in thousands) for processing a request and generating a 1kbyte reply from a cached static file.

Table 4.9 shows a comparison of the processing overhead of the user-level proxy approach with the implementation with kernel level support. The experimental settings for this comparison are not perfectly identical. While both schemes were implemented on the same hardware, the user-level proxy approach runs under the Solaris operating system and could not be easily ported to Linux due to a difference in the semantics of raw sockets. In addition, the server programs are different although they do similar processing. However, despite the experiment setting differences, the difference of almost a factor of five is clearly due mostly to the difference in the implementation of the scheme, not to OS differences. The large overhead of the proxy approach is caused by the extraneous system calls and message copying that are necessary for moving the messages between the two levels of proxies and the server.

4.6.5. Impact of Processor and Network Speed on System Performance

To evaluate the impact of processor on the performance of CoRAL, we ran some experiments on 2.66GHz Intel Pentium 4 Xeon PC's. Some of these experiments also evaluated the impact of a faster network interconnecting the two server replicas. For these latter experiments, the server hosts were interconnected by a Gb/sec switched network. The clients were identical to those of previous

experiments, and server replies were generated dynamically using the WebStone 2.0 CGI benchmark. The connections to the clients were always through a 100Mb/sec switched ethernet. In the rest of this section we present the results from the experiments on Pentium IV PC's and compare them with Pentium II results which were presented in the previous sections.

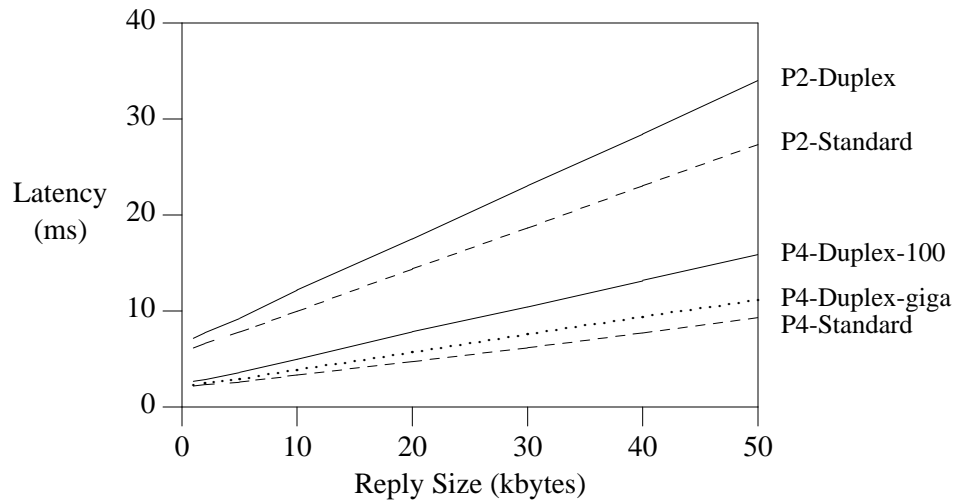


Figure 4.23: Average latency (ms) observed by a client for Pentium II (P2) and Pentium IV (P4) server hosts. P4-Duplex-100 and P4-Duplex-giga denote P4 system in duplex mode with 100 Mbit/sec and gigabit/sec network configuration respectively.

4.6.5.1. Latency

Figure 4.23 shows request-reply latencies, as measured on the clients, for Pentium II (P2) and Pentium 4 (P4) server hosts. The latencies for the P4 hosts with the Gb network fit Equation 2, with $\gamma = 0.0518 \mu\text{s/B}$, and for (standard, duplex), respectively, $\alpha = (0.0856, 0.102) \mu\text{s/B}$, $\beta = (1.67, 2.09) \text{ms}$. The figure shows that, for large replies, the latency overhead of duplex with 100Mb/sec inter-server connection relative to standard is much larger than the overhead of duplex

with the 1Gb/sec connection. As discussed earlier, most of the overhead for duplex is for logging the replies. By using a faster inter-server connection, the latency of logging the replies and thus the overhead are reduced significantly. The overhead for P4 servers with 100Mb/sec inter-server connections are slightly lower than the overhead for P2 servers, due to the use of a faster CPU.

4.6.5.2. Throughput

With the P2 servers, the duplex configuration reached a peak throughput of 1.8MB/sec (Figure 4.10). As shown in Figure 4.24, the P4 reach a peak throughput that is significantly higher: 6.0MB/sec and 7.8MB/sec with the 100Mb/sec and 1Gb/sec inter-server connection, respectively. With the P2 servers and with the P4 servers and Gb connection, the throughput is CPU-limited. On the primary, the logging of the reply to the backup and the sending of the reply to the clients occur over the same physical link. Hence, with the P4 servers and 100Mb connection, network bandwidth is the bottleneck. In this case, the reply throughput to the clients cannot exceed half the primary's outgoing link's throughput (50 Mbit/sec or 6.25 Mbytes/sec). Our results show that this bottleneck can be avoided with the faster inter-server connection.

4.6.6. Performance Under Overload

The logging of replies from the primary to the backup can result in undesirable performance characteristics under high load. Specifically, we discovered such a problem in our initial evaluation of the system with the server replicas operating in dual-role mode (Subsection 4.6.2.2.4) generating dynamic

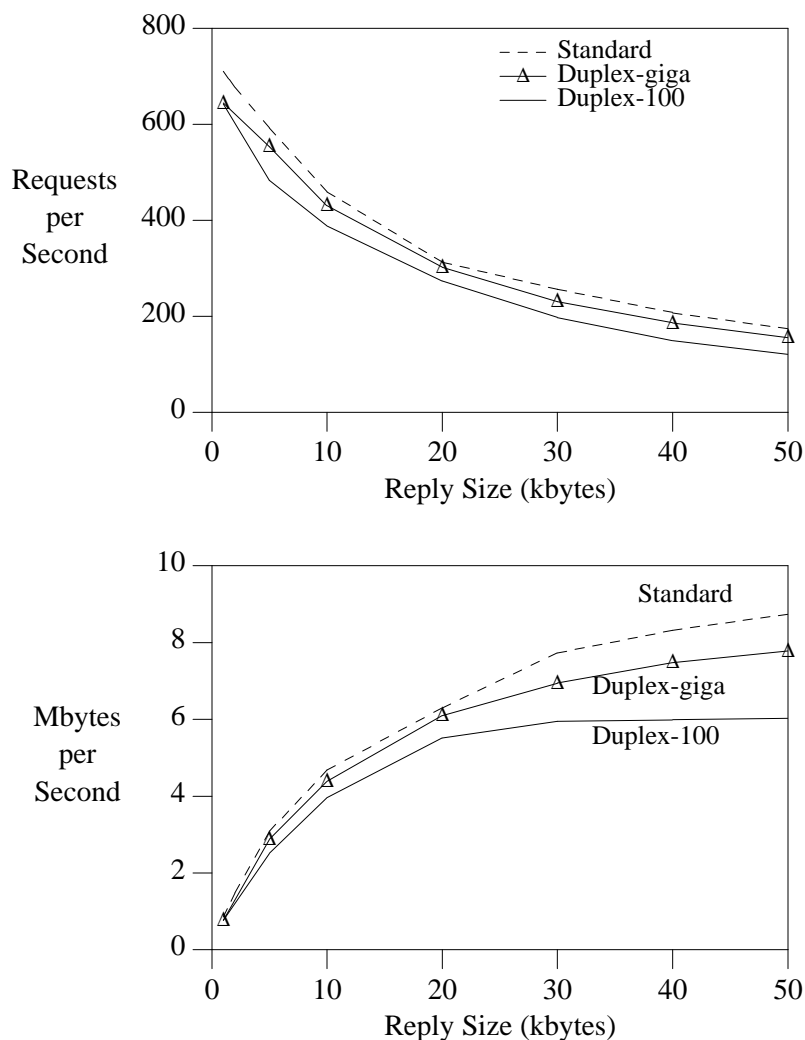


Figure 4.24: Peak system throughput for Pentium IV server hosts. Duplex-100 and Duplex-giga denote system in duplex mode with 100 Mbit/sec and gigabit/sec network configuration respectively.

replies. Starting from a low client request rate, as the request rate was increased, the throughput of the system increased. However, when the request rate exceeded the peak system throughput, there was a significant *decrease* in the achieved system throughput.

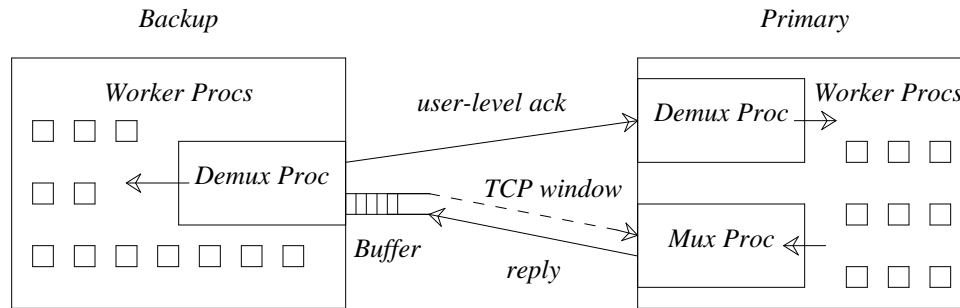


Figure 4.25: Performance degradation due to overload: If the backup becomes overloaded, the backup’s TCP receive buffer for the reply logging connection may become full. The unavailability of buffer space causes advertisement of small windows to the primary, resulting in decreased logging throughput.

In dual-role configuration, each server node acts as both primary and backup for different connections. Hence, if a server node experiences overload due to the processing of multiple requests as primary, its performance as backup is also effected. If a backup server is overloaded, the backup demux process (Subsection 4.4.2.4) is not scheduled frequently and the kernel receive buffer for the logging connection may become full (Figure 4.25). Since the logging is done over a TCP connection, this condition will cause the backup kernel TCP stack to reduce the TCP advertised window sent to the primary. The small (sometimes zero) window advertised by the backup causes the primary to reduce the send rate on the logging connection. This throughput reduction on the logging connection leads to the throughput reduction of the entire system.

Our solution to this problem was to give higher priority to the processing of messages received over the logging connection. This was done by assigning the mux and demux processes high priority using the standard Linux mechanism (`sched_setscheduler`) for assigning soft real-time priority to time-critical processes.

This simple change eliminated the anomalous performance characteristic we observed initially with no negative impact.

4.7. Fault Tolerance Validation Using Fault Injection

In order to validate the correctness of fault tolerance schemes, they must be tested under faults. This is often done by intentionally emulating (injecting) faults in some components of the system and monitoring their impact [Hsue97]. One of the basic tests of our system has been to physically disconnect one of the server hosts from the network. This action emulates a fail-stop host failure since that host immediately stops producing any visible output. This experiment was conducted over 50 times, and the system detected and recovered from the host failures correctly every time. The rest of this section describes experiments used to validate system operation under more likely fault scenarios.

We used a kernel-based software fault injector under development at the UCLA Concurrent Systems Laboratory. This injector emulates hardware faults by randomly flipping bits in CPU registers or memory of a process selected randomly out of a designated set of processes. The actual injection of faults is performed by a Linux kernel module. A user-level fault injector process periodically interacts with the kernel module using an `ioctl` call, and signals it to inject a fault into a target application process. After a fault is injected, it *may* cause an error, i.e., modify the behavior of a process, the next time the targeted process is scheduled to run by the OS. Some of the configurable parameters of the fault injector are: the frequency of injection, whether the injection is to registers or memory, and whether the injection is to the entire address space of the process or only to part of it (e.g.,

only to the stack segment). The fault injector we used does not inject faults into kernel registers or memory. Hence, faults are only injected into the state of user-level processes.

The fault injection experiments were conducted on 2.66GHz Intel Pentium 4 Xeon PC's interconnected by a gigabit/second switched network based on a D-Link DGS-1008D switch. Faults were injected into the primary node of CoRAL operating in duplex mode. The workload consisted of clients sending requests for 1KB replies, for a total reply throughput of 64.8 KB/sec. The clients compared the replies to known correct reply contents in order to determine whether the reply was corrupted. With each experiment, we monitored the system for recovery actions and inspected the replies received by clients for corruption.

We conducted targeted fault injection experiments, flipping bits in registers, or specific areas of memory. The fault injector was configured to randomly choose a user-level CoRAL process (either Apache worker processes or mux/demux processes) as the target for each injection. When a fault caused a server or process crash, a failover occurred and CoRAL processed requests in simplex mode. The system was configured to restore itself to duplex operation. A new server joined the simplex node 10 seconds after each failover, thus new faults could be injected into the system, allowing the experiments to run mostly unattended.

The server reintegration is implemented using two layers of shell scripts. The inner script starts a new initialized server (configured to join a simplex server) and does not return (sleeps forever). The outer script consists of a loop which 1) calls the inner script, 2) waits for it to return, and 3) waits an additional 10 seconds before going back to the top of the loop. Hence, a new initialized server is started

10 seconds after the inner script returns. The inner script must return whenever there is a crash in order to achieve the desired experiment setting. Hence, CoRAL is configured to kill the inner script as part of the conversion of process crashes to host crashes (Section 4.4.3). As a result, whenever a process crashes due to a fault injection, all processes and the inner script are killed, and a new server is initialized and joins the surviving simplex replica after 10 seconds.

Most faults injected into the system either had no effect or caused a process to crash. This result is consistent with fault injection experiments performed on other systems [Made02]. For each experiment we collected the number of injections, number of detected crashes, location of last injection prior to the crash (an indication of which fault injection caused the crash), and the number of corrupted/incorrect replies received by the clients.

4.7.1. Register Fault Injections

For the register fault injection experiments, the fault injector was configured to inject faults, at a rate of one fault every five seconds, into a process that was randomly selected out of all the processes related to the service, i.e., Web server processes, mux/demux processes, and the heartbeat generator and monitor. For each injection, a CPU register was randomly selected and a randomly chosen bit within that register was flipped.

Table 4.10 shows the results for register fault injections. For each register, the number of fault injections and host crash failures are shown. As expected, injections into critical registers such as the program counter (EPI) and segment registers cause frequent crash failures. In addition to the crash failures shown, 1%

Register	Crash Failures	Faults Injected	Failure Percentage
EBX	0	61	0
ECX	0	57	0
EDX	0	76	0
ESI	0	74	0
EDI	0	68	0
EBP	50	56	89
EAX	0	72	0
DS	43	50	86
ES	46	64	72
FS	0	76	0
GS	54	57	95
EIP	61	61	100
CS	54	60	90
EFL	3	58	5
UESP	57	57	100
SS	55	64	86
TOTAL:	423	1011	42

TABLE 4.10: CoRAL Fault Injection Experiment - Register Injections.

In addition to the crash failures shown, ten injections caused the service to hang.

of injections caused the system to hang. Since these faults did not cause host or process crashes, the system was not protected from them despite the CoRAL mechanism.

We did not observe any errors or failures due to faults injections into the general purpose registers. The web server behavior and implementation combined with limitations of the fault injector may be the cause this observed result. The web server implementation uses many wrappers around blocking system calls. The fault injector injects faults into processes that are not running, i.e., descheduled.

Hence, if a web server process is descheduled due to a blocking system call in a wrapper, faults injected into general purpose registers will not have any effects because they are immediately overwritten when the wrapper function returns.

In summary, out of 1011 injections, 57% had no impact, 42% caused a process crash, and 1% caused the system to hang. All crashes were handled correctly and full duplex operation was restored.

4.7.2. Memory Fault Injection

Faults were injected by flipping a random bit in a randomly-selected address in the address space of a process. As with register fault injection, the process was randomly selected out of all the processes related to the service. Table 4.11 shows the memory fault injection experiment results. When faults were injected into a randomly selected location within the entire process address space, errors occurred very rarely. Specifically, we observed only one failure (an incorrect reply received by a client) from more than 70000 fault injections. This is because our implementation uses a large (30MB) shared segment for communication between the Web server processes and the mux, demux, and heartbeat processes. With light load, most of this segment is unused. Hence, injecting faults into this segment very rarely has any impact.

In order to evaluate the behavior of our system when faults do cause errors, we performed two “stress tests,” where the injection was focused on memory areas that are more likely to cause errors.

In the first experiment, the fault injector was configured to exclude 98% of memory and inject faults only into memory that was not allocated using the *mmap*

Injection Target	Injections	Crashes	Bad Replies	Hangs
Entire process address space	70000	0	1	0
Non-mmap'ed memory(2%)	10358	84	1	3
Stack (typically 304-9588 bytes)	10114	211	0	5

TABLE 4.11: CoRAL Fault Injection Experiment - Memory Injections

system call. The mmap memory typically consists of large chunks, and in our case included memory used to store requests and replies. Hence, injections into those areas have a lower probability of causing crash failures. In this experiment, there was a crash roughly once in every 123 injections. In addition, in over 10,000 injections, one fault caused a client to receive a corrupted reply and three faults caused the system to hang and not process incoming client requests.

In the second experiment, the fault injector was configured to inject faults only into the stack. Since the stack typically contains some critical process state, as expected, a larger fraction of injections had an impact compared with our other memory injection experiments (see Table 4.11). However, even in this case, only two percent of the faults had an impact.

In all experiments, full recovery was achieved from all the crashes. Although our solution cannot handle failures that do not cause crashes (e.g., hangs or data corruption), in practice, most faults either cause crash failures or they do not cause any observable errors in the system. This conclusion is consistent with other fault injection research [Made02]. Hence, CoRAL can be expected to fully recover from the majority of errors that occur in practice.

4.8. Summary

We have developed a client-transparent fault tolerance scheme for Web services, called CoRAL, that correctly handles client requests, including those in progress at the time of failure, in spite of a Web server failure. For each connection, CoRAL uses two server hosts: a primary and a backup. TCP connection state is actively replicated while requests and replies are logged at the backup. If a server host fails, all connections transparently fail over to the backup. The system has the ability to restore a fault-tolerant configuration for new connections with a new host while existing connections remain active. From the perspective of the clients, partially received requests, requests being processed, and partially transmitted replies, all continue seamlessly despite server host failure.

We have implemented CoRAL with a standard Apache Web server running on Linux servers. Our implementation is based on a kernel module and a module for the Apache Web server. The kernel module is reusable for other TCP-based services and the Apache module code would be largely reusable for other request-reply protocols. The overhead of the full CoRAL mechanism involves some additional processing when connections are established plus the cost of logging replies to the backup. Optimizations for static content and the dual-role deployment of server hosts can minimize the impact of this overhead. Performance evaluation results show that in terms of latency, the overhead is too small to be noticeable for clients connecting over the Internet, even with very slow (350MHz) server hosts. Failover times on the order of 100-200ms are easily achieved. In practice, for the likely target applications of this scheme, replies are often small, dynamically generated, and require significant processing. For such workloads, the

results show that the overhead in terms of processing cycles, and thus maximum server throughput, is likely to be under 30%. Preliminary fault injection experiments demonstrate that the system will properly recover from the vast majority of faults.

Chapter Five

Transparent Fault-Tolerant Video Conferencing

Most of the existing work on client-transparent fault-tolerant network services has focused on web service [Aghd01, Aghd02, Aghd03a, Aghd03b, Alvi01, Koch03, Marw03, Shen00, Zhan04]. Hence, this work has generally dealt with TCP connections, HTTP transactions, and web servers. In this chapter, we examine how the methodology and mechanisms developed in the context of web service can be adapted and applied to a very different type of service — video conferencing. Video conferencing is an interesting test-case for applying fault tolerance for other types of services since it combines critical conference state that must be protected with media streams where limited data losses are acceptable. Furthermore, the application sensitivity to latency and high demands in terms of throughput and processing make video conferencing an interesting example for evaluation of a fault-tolerance solution.

The rest of this chapter is organized as follows. An introduction to off-the-shelf video conferencing applications is presented in Section 5.1. Section 5.2 discusses the details specific to the design and implementation of our fault-tolerant video conferencing scheme using OpenMCU [Quic03], an open source H.323 [Inte03] conferencing server developed on top of the OpenH323 library [Quic03]. Performance evaluation results are presented in Section 5.3. Validation of our scheme using fault injection experiments is presented in Section 5.4.

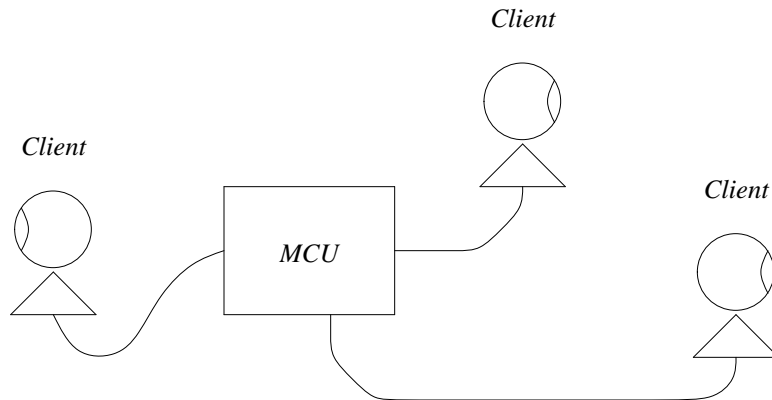


Figure 5.1: Video Conferencing with a Multi-Conferencing Unit (MCU).
An MCU fault can cause the conference to fail.

5.1. Introduction to Off-the-Shelf Video Conferencing

Off-the-shelf video conferencing schemes typically consist of multiple clients and a Multi-Conferencing Unit (MCU). The MCU is the central host for the conference (Figure 5.1). It executes the conferencing application and maintains the state of the conference. Clients connect to the MCU and join a conference with other clients. The main functionality of the MCU application is to route the multimedia stream (i.e. video) from each client to others. The MCU application may also provide additional functionality such as the mixing of multiple media streams, echo cancellation, content-adaptation to support different types of clients, and conference management such as maintaining multiple virtual rooms. Since conference state is kept only at the centralized MCU, an MCU process failure results in the failure of all active conferences handled by that MCU.

The conference state kept at the MCU must persist for the lifetime of a conference. This state changes during the conference as new clients join, some clients depart, etc. In order to achieve fault-tolerance any changes to the MCU

state must be recorded reliably and preserved across faults. The conference state changes infrequently and most of the MCU processing requirement is due to the processing of the media streams, where absolute reliability is not required to maintain the correctness of service. Hence, our approach to adding fault tolerance to the MCU combines fully replicated operation for maintaining critical conference state with a very low overhead failover mechanism for handling the media streams themselves.

5.2. Adding Fault Tolerance to a Video Conferencing Server

We used the same methodology discussed in Chapters 3 and 4 to add fault tolerance features to a video conferencing server. Our video conferencing implementation is based on OpenMCU [Quic03], an open source H.323 [Inte03] conferencing server developed on top of the OpenH323 library [Quic03]. The OpenMCU application maintains virtual conference rooms, receives and mixes media from the clients in a conference, and periodically transmits the mixed media to each client. TCP connections are used as the reliable channel for transmission of control messages. The media is transferred on top of unreliable channels using the UDP protocol.

As mentioned previously, the service identity, connection state, and relevant application state must be preserved over faults. We preserve the required state on a replica on the same LAN subnet. During fault-free (duplex) operation, the MCU connection state is actively replicated and the application state is synchronized as necessary in order to handle non-deterministic behavior. If one of the replicas fails, the surviving MCU takes over and operates in simplex mode. Our

implementation requires changes to the MCU at OS and application levels. A kernel module on each replica facilitates the functionality necessary for replicating the connection state. Modifications to the application code are used to implement application level synchronization and fault detection.

Our implementation consists of roughly 5000 lines of user-level code and also 5000 lines of kernel-level code in the form of a kernel module. We also had to modify the existing application code (less than 50 lines) and kernel code (less than 100 lines). Most of the modifications created entry points for the rest of our code. Our user-level code consists of less than 1000 lines of application specific code whose main functionality is the synchronization of application-level state. This code was written specifically for the MCU application and cannot be used for other types of services. The rest of user-level code, which mainly deals with fault-detection and failover, is mostly application independent and very similar to the code used for the CoRAL fault-tolerant web service (Chapter 4). The kernel-level code is not application specific. The TCP portion is exactly the code that was used for CoRAL, and the UDP code can be used with other services that use UDP connections.

Some aspects of our fault-tolerant video conferencing implementation are directly reused from the kernel-based implementation of CoRAL (Section 4.4.2) or are a minor adaptation (e.g., initialized with a different TCP port number). The shared aspects include: 1) preservation of reliable communication across failures, i.e., active replication of TCP connections (Section 4.4.2.1) for the video conferencing control channel, 2) error detection and failover (Section 4.4.4), and 3) conversion of process crashes to host crashes (Section 4.4.3). The new aspects of

the scheme developed specifically for video conferencing are: 1) preservation of unreliable communication (i.e., UDP media connections) across failures, and 2) replication of application state and handling of application-level non-determinism. In the rest of this section, we will discuss the aspects of the implementation that are different from CoRAL's implementation.

5.2.1. Unreliable Communication

The MCU media channel is built on top of UDP. In order to preserve the media channel across faults, identical UDP socket structures must be created on both MCU replicas. In addition, client UDP packets sent to the faulty MCU must be transparently routed to the surviving replica after a fault. H.323 services typically create the media stream after receiving a client request on the control channel. With OpenMCU, UDP sockets are created in response to client requests received on the TCP connection. Since we replicate the TCP connection, we are ensured that both replicas will receive identical requests. Hence, if identical applications are executed on both replicas and the application behavior is deterministic, or non-determinism is handled as shown later in this section, then identical UDP socket structures will be created on both replicas.

The key to the low overhead of our scheme is that the UDP packets (the media streams) are not really processed by the backup. The handling of UDP packets by replicas can be implemented using several configurations. One approach is the use of forwarding similar to our TCP replication. In fault-free operation, the service IP address for the media stream is mapped to the backup. As a result, client packets arrive at the backup first. Upon the receipt of client UDP

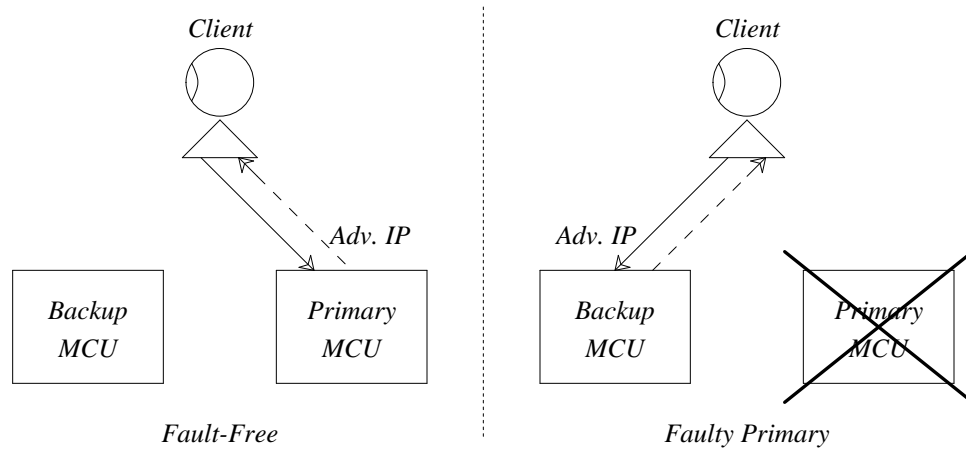


Figure 5.2: Direct UDP communication between client and primary. Advertised IP address is mapped to the primary in fault-free operation. If primary MCU fails, the backup takes over the advertised address and continues operation in simplex mode.

packets, the backup forwards a copy to the primary by changing the destination IP address in the packet. Outgoing packets are transmitted only by the primary, with the source IP address always being set to the advertised address to maintain transparency. Faults may occur on either primary or backup replicas. If the primary MCU fails, the backup no longer forwards packets to the primary, and it starts transmitting outgoing packets to clients. If the backup MCU fails, the primary takes over the identity of backup server and client packets arrive directly at the primary.

Another approach is direct UDP communication between clients and the primary (Figure 5.2). During fault-free operation, the service IP address for the media stream is mapped to the primary and UDP packets are exchanged directly between the client and primary MCU. If the primary MCU fails, the backup takes over the service IP address, receives incoming client packets, and transmits

outgoing packets to clients. If the backup MCU fails, the communication between the clients and primary can continue normally, albeit now without fault-tolerance features. The direct client to primary communication eliminates the forwarding overhead that is incurred with the forwarding approach. However, since our TCP replication implementation maps the service address to the backup in fault-free operation, this approach requires the decoupling of reliable and unreliable connections at the application protocol level. The H.323 protocol includes this features. The advertised address of the unreliable channel is given to the client by the server over the control channel. Hence, the service IP address for the control and media channels need not be the same. However, this feature is not commonly used. In fact theopenh323 library implementation assumes that the two addresses will always be identical. We made a small modification to the library, allowing the use of different IP addresses for the TCP and UDP connections.

A third approach is to use IP multicast. An IP multicast address is used as the advertised address of the media stream. Hence, the network multicasts the client UDP packets to both replicas (Figure 5.3). In fault-free operation, only the primary transmits outgoing packets to the client. If the primary fails, the backup starts transmitting outgoing packets. If the backup fails, no changes take place at the primary. The use of global IP multicast addresses for such small scale multicast may not be practical, especially for services provided over the Internet. As an alternative, a server-side node or router can be used to multicast client UDP packets and achieve the same effect. The advertised IP address for the media is mapped to the multicast node. The multicast node simply sends a copy of each client packet that it receives to both replicas. The fault-free and failover operation

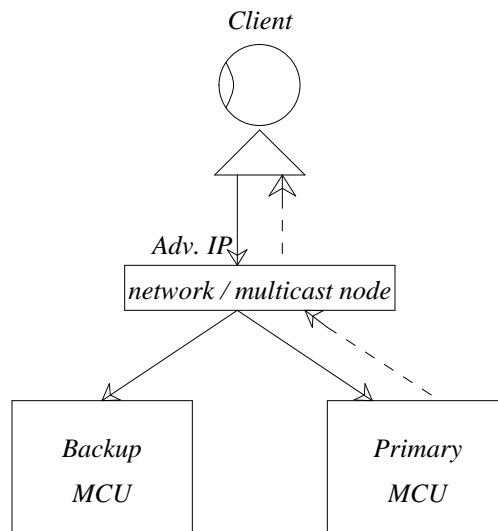


Figure 5.3: UDP communication using IP multicast or multicast node. Client UDP packets are multicast to both replicas by either the network (IP multicast) or a multicast node. The advertised address is an IP multicast address or address of multicast node, respectively. In fault-free operation only the primary replica sends outgoing packets to the client.

of the replicas are identical with using an IP multicast address. The drawback is that the multicast node introduces a new point of failure, although in practice it is less likely to fail due to its simple behavior. Since the multicast node is stateless — it does not contain any connection or application state — it is relatively easy to make it fault tolerant. All that is required is a backup that detects faults and simply takes over the IP address and continues the processing of packets in the same manner.

5.2.2. Application State and Non-determinism

The MCU application can be actively replicated, with identical copies available and running on both replicas. Since the application input, i.e., TCP and UDP connections, are replicated, the application state will also be identical if the processing is deterministic. The OpenMCU application is not deterministic, hence the applications on the primary and backup must be synchronized whenever there is a non-deterministic state change. We found only a few such events: the selection of initial sequence number and SyncSourceOut variable used by the RTP connection, and creation of the UDP port used for media transfer. Fortunately, they all occur at the creation and initialization of each RTP session and therefore can be synchronized together. We synchronize the replicas by exchanging messages. The primary sends the non-deterministic state changes to the backup. The backup makes state changes according to the primary's message — it sets initial values for variables and creates UDP ports — and sends an acknowledgment back to the primary. The primary waits for the acknowledgment message before it continues. Hence, the application states will be identical before the RTP session is used. If the primary fails during the synchronization procedure before the synchronization message reaches the backup, the backup can safely make its own non-deterministic state changes because the primary state changes will not have been visible to the client.

If external processes are allowed to use resources on the replica hosts, synchronization of non-deterministic state changes as described above may not be possible. For example, the backup may not be able to create a UDP socket with the same port number as the primary because another process is using it. To get

around such resource conflicts, we added the possibility of a negative acknowledgment response to the synchronization messages. If the backup cannot make the required state changes, it sends a nack messages back to the primary, informing it to undo its state changes and try a different path. In the UDP example, the primary deallocates the socket structure, creates a new socket bound to a different port number, and retries synchronization with the backup.

Active replication of MCU application inherently incurs a large performance overhead. The backup application performs every operation performed by the primary. Some application operations may not affect the application state and are not required to be performed by a replica if the only goal is to preserve the application state. For example, the media processing performed by the MCU is stateless. During fault-free operation, it is not necessary for the backup MCU to process or generate media. Only the control stream which affects the application state must be processed. Hence, as an optimization, we modified OpenMCU and disabled all operations related to the media stream in backup mode while keeping the control operations intact. Our evaluation results in Section 5.3 show that we successfully eliminated almost all the processing on the backup.

5.3. Performance Evaluation

Evaluation experiments were performed on 2.6 GHz Intel Pentium IV Xeon PC's interconnected by a gigabit/second switched network based on a D-Link DGS-1008D Switch. The MCU nodes were running our modified OpenMCU application on top of the Linux 2.4.20 kernel with our kernel module installed. We used three client nodes, each running the ohphone [Quic03] application with a

Logitech Quickcam Pro 3000 webcam. The webcams were pointed towards monitors where the screen refresh (i.e., flicker) created a constantly changing image. Finally, the three clients joined a conference hosted on the MCU.

Faults were injected into the system by physically disconnecting a MCU from the network (i.e., a host crash), and by killing the MCU application processes with the kill command (i.e., a process crash). Our implementation successfully recovered from both types of faults. Qualitatively, with moving video, faults cause a brief interruption of video at each client, with the magnitude of duration being in the order of the heartbeat period. If the video is static and not moving, there is a noticeable (few seconds) impact on the image sent to the clients. The reason is that the replica taking over (i.e. backup) starts with a blank image. The video image is partitioned into multiple chunks and the client codec transmits the static parts of the video image less frequently. Hence, a few seconds are required for the backup to receive the entire image. Fortunately, this has essentially no impact on the ability to continue the conference since audio does not suffer from this problem. Specifically, the audio codec does not rely on history. Hence, the audio interruption duration is always on the order of the heartbeat period, and is barely noticeable with typical (<100msec) heartbeat settings.

5.3.1. Processing Overhead

We measured the processing cycles used by the MCU using the CPU's performance monitoring registers and Pettersson's Performance-Monitoring Counters Driver [Pett03]. We measured `global_power_events` [Inte04] which accumulates the time during which the processor is not stopped. CPU cycles used

by a bare system were deducted to derive the actual cycles used by the MCU. For this experiment, the heartbeat rate was set at 100 milliseconds, and the system throughput was varied by configuring the clients and MCU to transmit video at different number of frames per second.

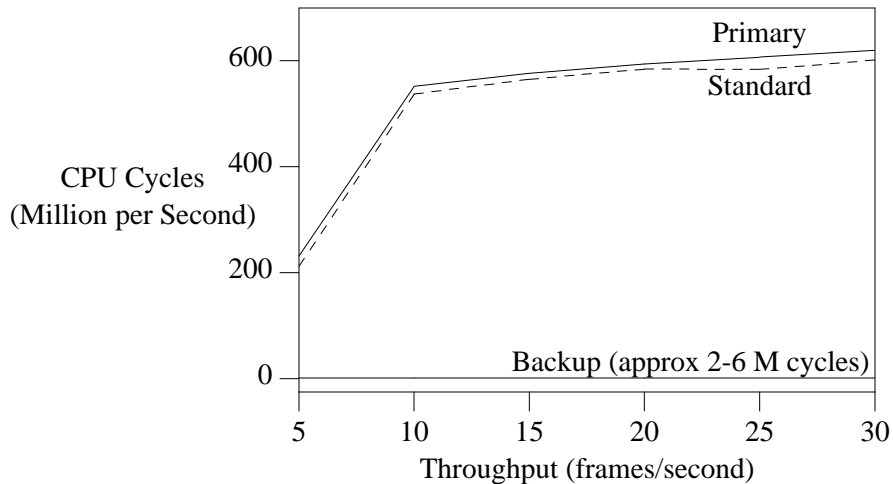


Figure 5.4: CPU cycles (million per second) used by a standard off-the-shelf MCU and the primary and backup MCU's in our scheme. System throughput was varied configuring clients and MCU to use video with different number of frames per second.

Figure 5.4 show the CPU cycles used by our primary and backup MCU and a standard off-the-shelf MCU without fault-tolerance features. The backup MCU is mostly idle. It uses approximately 2 to 6 million cycles per second depending on the UDP connection handling (discussed below). The primary consumes slightly more cycles than a standard server mainly due to bookkeeping operations in the kernel module and heartbeat generation and monitoring. Our overall processing overhead, the difference between the primary and standard cycles plus the cycles used by the backup, is small — roughly 3 percent for the direct UDP scheme at 30

frames per second.

The idle cycles on the backup MCU node are wasted if not used. Those cycles can obviously be used for processing of other applications. It is also possible to deploy the fault-tolerant service in dual-role (Subsection 4.5.1) fashion, where each MCU acts as both a primary and backup for different conferences. Hence, the idle backup cycles are used by another instance of the MCU acting as the primary.

5.3.2. Failover Latency

When a MCU fails, the system is briefly unavailable while a failover takes place. The failover time includes the time to detect a fault (i.e., consecutive missed heartbeats) and the time for the reconfiguration of system from duplex to simplex mode. During failover client packets sent to the MCU may be lost resulting in a visible interruption to the clients. We quantified the interruption by measuring the lost client packets during failover, and approximating the failover time based on that information. In this experiment faults were injected into the system by disconnecting one of the MCU replicas from the network.

Figure 5.5 shows the interruption time due to failover caused by primary or backup MCU faults. The heartbeat rate has a major effect on the length of interruption, showing that failover time is dominated by the fault detection time. Primary faults cause an interruption in all schemes. The forwarding and multicast-node schemes' failover times are virtually identical. The direct scheme has a slightly higher failover time because the backup must takeover the primary's IP address which is used as the identity of the UDP connection. The other two

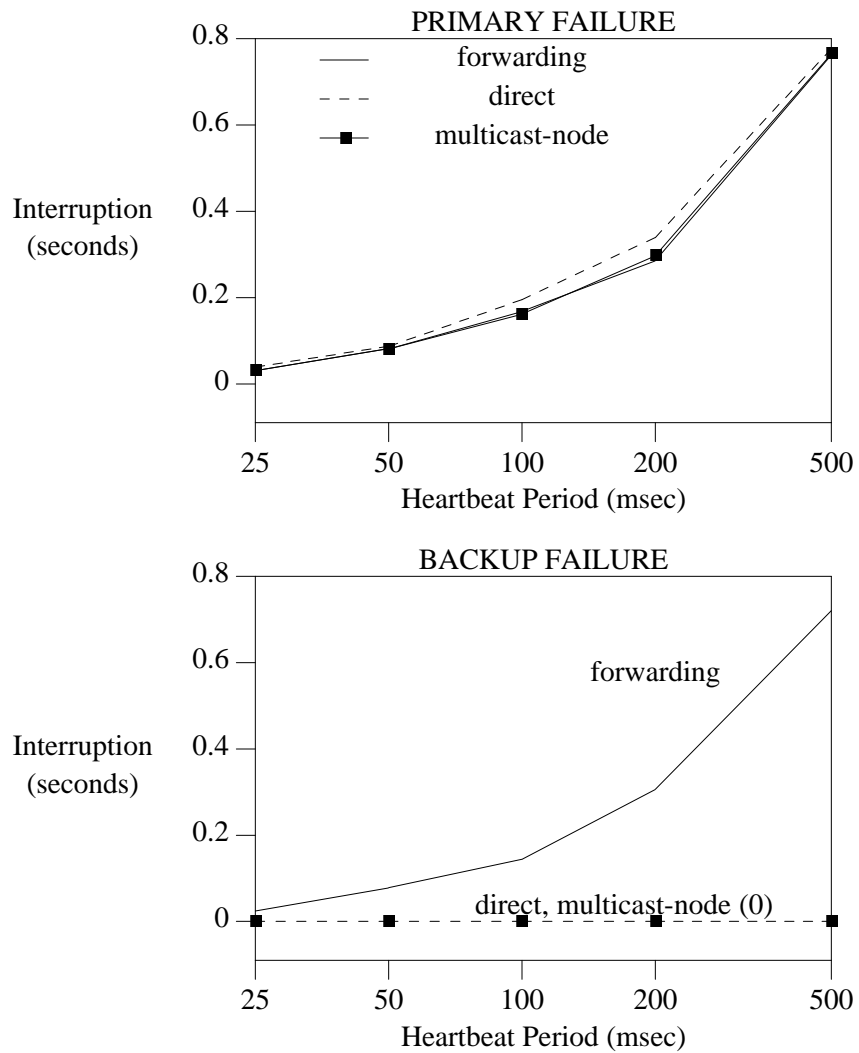


Figure 5.5: Interruption time due to failover for different UDP schemes.

schemes do not require an address takeover if the primary fails.

Backup faults cause an interruption only in the forwarding scheme. If the backup fails with the forwarding scheme, client UDP packets will not reach the primary until the fault is detected and a failover occurs. With forwarding scheme, backup faults cause slightly shorter interruption than primary faults (shown on

different sub figures) because IP address takeover is not necessary for a backup fault. Backup faults in the direct and multicast-node schemes do not cause an interruption because client UDP packets are not lost. The primary MCU detects the fault and configures itself for simplex operation without interrupting the media stream. The reconfiguration from primary to simplex is necessary only for TCP connection failover and fault detection processes.

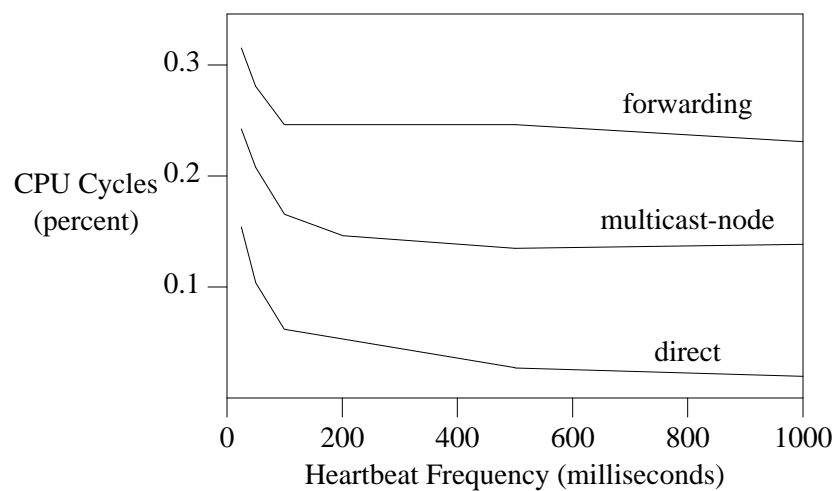


Figure 5.6: Percent of available CPU cycles used by the backup MCU in duplex mode for different UDP configurations. Experiments were run on 2.6 GHz PCs. Throughput was kept constant at 30 frames per second.

5.3.3. Impact of Heartbeat Rate and UDP Configuration Choice

Figure 5.6 shows the percentage of available processing cycles used by the backup MCU with different heartbeat rates. The system throughput was kept constant at 30 frames per second. Overall, the backup MCU is mostly idle. Each scheme uses fewer than one percent of the available CPU cycles on our 2.6 GHz

machines. The direct scheme is the most efficient, using the fewest cycles. The multicast-node scheme has more processing overhead than the direct scheme because the backup nodes receive a copy of every client UDP packet. Cycles are used up when these packets are received and copied to a buffer, examined, and then discarded. The forwarding scheme has the most overhead because every client packet is copied, its header rewritten, and then forwarded to the primary. The heartbeat rate of course has an effect on the amount of processing cycles used for all the schemes. Since the heartbeat rate also affects the failover time, there is a tradeoff between processing overhead and faster failover time.

5.3.4. TCP Replication and Application Synchronization

Most of the MCU processing requirement is due to the processing of the media streams. Hence, the evaluation results above are focused on the media streams and UDP connections. However, there is also overhead due to TCP connection replication and application synchronization. To join a conference, a client establishes a TCP connection and sends a control message. With our scheme, the TCP connection is replicated and relevant application state is synchronized between the replicas at join time. We measured the CPU cycles used for the establishment of the control channel and the addition of a client to a conference. Each replica consumes roughly 155 million CPU cycles for the connection establishment and application synchronization. This initialization cost is negligible compared to the millions of cycles used *every* second for the media processing. Other control messages (e.g., connection close, change in throughput) may also be exchanged between the MCU and clients, but they are typically

infrequent.

5.4. Fault Tolerance Validation Using Fault Injection

Similar to the CoRAL fault injection experiments discussed in Section 4.7, we also conducted fault injection experiments on our fault-tolerant video conferencing scheme to validate and evaluate our implementation. The fault injection experiments were performed with the system operating in duplex mode with three clients (webcams) participating in an active conference.

First we tried the simple experiment where one of the MCU replicas was physically disconnected from the network. This experiment was repeated dozens of times, with our implementation successfully recovering each time. The surviving replica detected the failure, a failover to simplex mode was performed, and the clients continued their conference with at most a brief interruption as mentioned in Section 5.3.

We also conducted software based fault injection experiments using the same kernel-based fault injector described in Section 4.7. As with the fault injection for the Web service (Subsection 4.7.1), the fault injector ran on the *primary* host and was configured to inject faults, at a rate of one every five seconds, into a process that was randomly selected out of all the processes related to the service, i.e., the MCU process (that consists of multiple threads), mux/demux processes, and the heartbeat generator and monitor. For each injection, a CPU register was randomly selected and a randomly chosen bit within that register was flipped. Table 5.1 shows the results for register fault injections. Our implementation successfully recovered from all of the crash failures. 27 percent of injections resulted in a crash

Register	Crash Failures	Faults Injected
EBX	1	11
ECX	0	5
EDX	0	9
ESI	1	8
EDI	1	11
EBP	4	8
EAX	0	12
DS	5	8
ES	3	7
FS	0	2
GS	3	8
EIP	5	13
CS	6	8
EFL	1	10
UESP	4	8
SS	3	8
TOTAL:	37	136

TABLE 5.1: Video Conferencing Fault Injection Experiment

failure that was detected and recovered from, with the rest having no visible effect on the system.

As discussed in Subsection 4.7.1, the effects of the fault injection become visible only when a blocked process is rescheduled. The application synchronization (mux/demux) processes are used infrequently with video conferencing — only when a new client arrives and joins a conference. With our experimental setup, no clients join or depart the conference during the experiment. Hence, in our experiment, injection to the mux/demux processes will have no effect. Therefore, it is expected that, in practice, a larger fraction of faults than

indicated by our results will cause crash failures.

While injecting register faults to the general-purpose registers of the web service had no impact, such injection into the video conferencing service (e.g., to the EBX register) did result in some crash failures (Table 5.1). The explanation lies in the differences in the application behavior (web server versus MCU) and their interaction with the fault injector. The MCU application is more CPU intensive than the web server. Therefore, MCU processes may be descheduled at any point in the execution, whenever their time quanta runs out. Hence, injections into general purpose registers may occur in midst of a function that was using them, thus incorrect register values become visible to the application and possibly cause a failure. On the other hand, as discussed in Subsection 4.7.1, the web server processes are almost always descheduled due to the execution of a blocking system call. Furthermore, the web server implementation includes function wrappers around blocking calls, resulting in a function return call and thus overwriting of general purpose registers upon the return from a blocked call. As a result, since the injection has an impact only when the process is rescheduled, injections into general purpose register of web server processes are almost always overwritten.

In summary, our video conferencing fault injection experiments show that in practice, faults either do not cause any observable errors in the system, or they cause a MCU process to crash. In all cases where a crash occurred, our scheme detected the failure, failover operations were performed transitioning the system to simplex operation, and the clients continued their active conferences without noticing that a failure had occurred.

5.5. Summary

We have presented the design and implementation of a client-transparent fault-tolerant video conferencing service that can recover from host or process crashes at the server site. We believe that this would be the first publication of a practical client-transparent fault tolerance scheme for video conferencing. Our implementation is based on an existing conferencing server (MCU) and required modification of a tiny fraction of the existing code. Most of the user-level and kernel-level code in our implementation is directly usable or easily adaptable for other network services. This is verified by the fact that the TCP components were minor adaptations of our previous work on fault-tolerant web service. Our measurements show that the processing overhead of our scheme during fault-free operation is insignificant — approximately 3% additional CPU cycles. Our experiments demonstrated successful recovery from all host crashes and process crashes, including process crashes caused by random fault injection to CPU registers. With a low-overhead configuration (heartbeat period of 25ms-50ms), when a fault occurs, the conference proceeds with no loss of audio (a barely noticeable “blip”) and a minor disruption of static video images that clears up in a few seconds.

Chapter Six

Summary and Conclusions

People are increasingly dependent on a wide variety of network services. As with other “utilities,” service providers will be expected to deliver “always on” highly-reliable service. In order to meet these expectations, extensive use of fault-tolerance will be required. Many network services already have a large number of deployed clients. In addition, service clients tend to be generic and developed independently of the service. Hence, unless standardized fault-tolerance protocols become widely adopted, it is imperative for fault-tolerance solutions to be client-transparent. With this motivation, we have developed a general methodology for engineering client-transparent fault-tolerant network services using commercial off-the-shelf components.

Most existing fault-tolerance solutions for network services do not provide fault-tolerance for active connections at failure time, expect servers to behave deterministically, or they require changes to the clients. These limitations are unacceptable for many current and future network service applications. Our solution, based on a hot standby backup approach, is transparent to the clients and requires minimal changes to the server OS and application.

Our proposed methodology for client-transparent fault-tolerant network services defines three key server-side service components that must be preserved across failures: service identity, connection state, and application state. We presented various possible techniques for preservation of these components and discussed the associated tradeoffs. We have evaluated our methodology by building, using off-the-shelf components, two fault-tolerant prototypes of popular

network services — web service and video conferencing.

CoRAL, our client-transparent fault-tolerance scheme for web services correctly handles all client requests in spite of a web server failure. The TCP connection state is actively replicated on a standby backup. At the application level, HTTP requests and replies are logged. In the event of a server failure, TCP connections transparently failover to the backup. The unsent portions of any logged replies are sent to the client, and requests for which a reply was not logged are reprocessed. Hence, all requests — including those being processed at failure time — are handled correctly. Performance evaluation results show that in terms of latency, the overhead is too small to be noticeable for clients connecting over the Internet, even with very slow (350MHz) server hosts. Failover times on the order of 100-200ms are easily achieved. In practice, for the likely target applications of this scheme, replies are often small, dynamically generated, and require significant processing. For such workloads, the results show that the overhead in terms of processing cycles, and thus maximum server throughput, is likely to be under 30%.

We built upon our experience with implementing fault-tolerant web service and adapted our general methodology for client-transparent fault-tolerant network service to another important network service — video conferencing. We have presented the design, implementation, and evaluation of a client-transparent fault-tolerant video conferencing service that can recover from host or process crashes at the server site. Our implementation is based on an existing conferencing server (MCU) and required modification of a tiny fraction of the existing code. Our measurements show that the processing overhead of our scheme during fault-free operation is insignificant — approximately 3% additional CPU cycles. With a

low-overhead configuration (heartbeat period of 25ms-50ms), when a fault occurs, the conference proceeds with no loss of audio (a barely noticeable “blip”) and a minor disruption of static video images that clears up in a few seconds.

Our methodology for constructing client-transparent fault-tolerant network services can be applied to other applications. Most of the user-level and kernel-level code in our implementation is directly usable or easily adaptable for other network services. This is verified by the fact that the TCP components for our video conferencing implementation were minor adaptations of CoRAL — our previous work on fault-tolerant web service. Our performance evaluation results indicate acceptable overheads, much lower than duplication, are achievable. Sub-second error detection and failover times of our scheme successfully mask faults from clients for most applications. Preliminary fault injection experiments on our web service and video conferencing implementations demonstrated that our scheme will properly recover from the vast majority of faults.

Bibliography

- [Aghd01] N. Aghdaie and Y. Tamir, "Client-Transparent Fault-Tolerant Web Service," *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, pp. 209-216 (April 2001).
- [Aghd02] N. Aghdaie and Y. Tamir, "Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support," *Proceedings of the The 11th International Conference on Computer Communications and Networks*, pp. 63-68 (October 2002).
- [Aghd03a] N. Aghdaie and Y. Tamir, "Performance Optimizations for Transparent Fault-Tolerant Web Service," *Proceedings of 2003 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 29-32 (August 2003).
- [Aghd03b] N. Aghdaie and Y. Tamir, "Fast Transparent Failover for Reliable Web Service," *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 757-762 (November 2003).
- [Aghd05] N. Aghdaie and Y. Tamir, "Efficient Client-Transparent Fault Tolerance for Video Conferencing," *Proceedings of the 3rd IASTED International Conference on Communications and Computer Networks*, (October 2005).

- [Alme98] J. Almeida and P. Cao, “Wisconsin Proxy Benchmark,” *Technical Report 1373, Computer Sciences Dept, Univ. of Wisconsin-Madison*, (April 1998).
- [Alvi01] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, “Wrapping Server-Side TCP to Mask Connection Failures,” *Proceedings of IEEE INFOCOM*, pp. 329-337 (April 2001).
- [Ande96] E. Anderson, D. Patterson, and E. Brewer, “The Magicrouter, an Application of Fast Packet Interposing,” *Class Report, UC Berkeley* - <http://www.cs.berkeley.edu/~eanders/projects/magicrouter/>, (May 1996).
- [Andr96] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra, “SWEB: Towards a Scalable World Wide Web Server on Multicomputers,” *Proceedings of the 10th International Parallel Processing Symposium*, pp. 850-856 (April 1996).
- [Apac98] Apache Software Foundation, “The Apache HTTP Server Project,” <http://httpd.apache.org/>, (1998).
- [Aver00] L. Aversa and A. Bestavros, “Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting,” *Proceedings of the 2000 IEEE International Performance, Computing, and Communications Conference*, pp. 24-29 (February 2000).
- [Aviz04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,”

- IEEE Transactions on Dependable and Secure Computing* **1** pp. 11-33 (January-March 2004).
- [Bail01] B. Bailey, "Acceptable Computer Response Times," *UI Design Update Newsletter* - <http://www.humanfactors.com/downloads/apr01.asp>, (April 2001).
- [Bern96] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0," RFC 1945, IETF (May 1996).
- [Best98] A. Bestavros, M. Crovella, J. Liu, and D. Martin, "Distributed Packet Rewriting and its Application to Scalable Server Architectures," *Proceedings of the International Conference on Network Protocols*, pp. 290-297 (October 1998).
- [Bilo03] R. Bilorusets, A. Bosworth, D. Box, F. Cabrera, D. Collison, J. Dart, D. Ferguson, and C. Ferris, *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*. March 2003.
- [Blac92] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, and R. P. Draves, "Microkernel Operating System Architecture and Mach," *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 11-30 (April 1992).
- [Bode95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro* **15**(1) pp. 29-36 (February 1995).
- [Borg83] A. Borg, J. Baumbach, and S. Glazer, "A Message System

- Supporting Fault Tolerance,” *9th Symposium on Operating Systems Principles*, pp. 90-99 (October 1983).
- [Bres99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, “Web Caching and Zipf-like Distributions: Evidence and Implications,” *Proceedings of IEEE INFOCOM*, (March 1999).
- [Bris95] T. Brisco, “DNS Support for Load Balancing,” RFC 1794, IETF (April 1995).
- [Burt02] N. Burton-Krahn, “HotSwap - Transparent Server Failover for Linux,” *Proceedings of USENIX LISA '02: Sixteenth Systems Administration Conference*, pp. 205-212 (November 2002).
- [Chu00] Y.-h. Chu, S. G. Rao, and H. Zhang, “A Case For End System Multicast,” *Proceedings of ACM SIGMETRICS*, pp. 1-12 (June 2000).
- [Chu01] Y.-h. Chu, S. G. Rao, S. Seshan, and H. Zhang, “Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture,” *Proceedings of ACM SIGCOMM*, pp. 55-67 (August 2001).
- [Cisc99a] Cisco Systems Inc, “Failover Configuration for LocalDirector,” *Cisco Systems White Paper* - http://www.ieng.com/warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm, (1999).
- [Cisc99b] Cisco Systems Inc, “Scaling the Internet Web Servers,” *Cisco Systems White Paper* -

http://www.ieng.com/warp/public/cc/pd/cxsr/400/tech/scale_wp.htm,
(1999).

- [Cunh95] C. Cunha, A. Bestavros, and M. Crovella, “Characteristics of World Wide Web Client-based Traces,” *Technical Report TR-95-010, Boston University, CS Dept, Boston, MA 02215*, (April 1995).
- [Dias96] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari, “A scalable and highly available web server,” *Proceedings of IEEE COMPCON '96*, pp. 85-92 (1996).
- [Evan03] C. Evans, D. Chappell, D. Bunting, and G. Tharakan, “Web Services Reliability (WS-Reliability),” <http://www.oasis-open.org/committees/wsrn/charter.php>, OASIS Web Services Reliable Messaging Technical Committee, Working Draft (January 2003).
- [Fiel99] R. Fielding, J. Mogul, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” RFC 2616, IETF (June 1999).
- [Frol00] S. Frolund and R. Guerraoui, “Implementing e-Transactions with Asynchronous Replication,” *IEEE International Conference on Dependable Systems and Networks*, pp. 449-458 (June 2000).
- [Golu90] D. Golub, R. Dean, A. Forin, and R. Rashid, “Unix as an Application Program,” *Proceedings of Summer USENIX Conference*, pp. 87-96 (June 1990).
- [Gray92] J. Gray and A. Reuter, *Transaction Processing : Concepts and*

- Techniques*, Morgan Kaufmann Publishers (September 1992).
- [Hsue97] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer* **30**(4) pp. 75-82 (April 1997).
- [IBM 2] IBM Inc, "HTTPR Specifications," <http://www-106.ibm.com/developerworks/webservices/library/ws-httpRSpec/>, (April 2002).
- [Inte00] Intel Inc, "Intel NetStructure e-Commerce Products," <http://www.intel.com/support/netstructure/commerce/index.htm>, (2000).
- [Inte04] Intel Inc, *IA-32 Intel Architecture Software Developer's Manual - Volume 3: System Programming Guide*. 2004.
- [Inte03] International Telecommunication Union, "Packet-based multimedia communications systems," *ITU-T, Recommendation H.323 Draft V5*, (May 2003).
- [John88] D. B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 171-181 (August 1988).
- [Katz94] E. D. Katz, M. Butler, and R. McGrath, "A Scalable HTTP Server: The NCSA Prototype," *Computer Networks and ISDN Systems* **27**(2) pp. 155-164 (1994).
- [Keis64] W. Keister, R. W. Ketchledge, and H. E. Vaughan, "No. 1 ESS

- System Organization and Objectives,” *Bell System Technical Journal* **43**(5) pp. 1831-1844 (September 1964).
- [Ketc65] R. Ketchledge, “The No. 1 Electronic Switching System,” *IEEE Transactions on Communications* **13** pp. 38-41 (March 1965).
- [Knig98] S. Knight, D. Weaver, D. Whipple, R. Hinden, D. Mitzel, P. Hunt, P. Higginson, M. Shand, and A. Lindem, “Virtual Router Redundancy Protocol,” RFC 2338, IETF (April 1998).
- [Koch03] R. Koch, S. Hortikar, L. Moser, and P. Melliar-Smith, “Transparent TCP Connection Failover,” *International Conference on Dependable Systems and Networks (DSN-2003)*, pp. 383-392 (June 2003).
- [Lamp82] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Transactions on Programming Languages and Systems* **4**(3) pp. 382-401 (July 1982).
- [Li98] T. Li, B. Cole, P. Morton, and D. Li, “Cisco Hot Standby Router Protocol (HSRP),” RFC 2281, IETF (March 1998).
- [Luo01] M.-Y. Luo and C.-S. Yang, “Constructing Zero-Loss Web Services,” *Proceedings of IEEE INFOCOM*, pp. 1781-1790 (April 2001).
- [Made02] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, “Experimental Evaluation of a COTS System for Space Applications,” *International Conference on Dependable Systems and Networks (DSN’02)*, pp. 325-330 (June 2002).

- [Marw03] M. Marwah, S. Mishra, and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," *International Conference on Dependable Systems and Networks (DSN-2003)*, pp. 373-382 (June 2003).
- [Matr00] Matrix NetSystems Inc, "The Internet Weather Report," <http://www.internetweather.com>, (2000).
- [Mind02] Mindcraft Inc, "WebStone Benchmark Information," <http://www.mindcraft.com/webstone>, (2002).
- [Orac99a] Oracle Inc, *Oracle8i Distributed Database Systems - Release 8.1.5*, Oracle Documentation Library (1999).
- [Orac99b] Oracle Inc, "Oracle Parallel Server: Solutions for Mission Critical Computing," *An Oracle Technical White Paper* - http://www.oracle.com/database/documents/parallel_server_twp.pdf, (February 1999).
- [Pett03] M. Pettersson, "Linux x86 Performance-Monitoring Counters Driver," <http://www.csd.uu.se/~mikpe/linux/perfctr/>, (2003).
- [Plum82] D. C. Plummer, "An Ethernet Address Resolution Protocol," RFC 826, IETF (November 1982).
- [Quic03] Quicknet Technologies Inc, "OpenH323 Project," <http://www.openh323.org>, (2003).
- [RedH01] RedHat Inc, "TUX Web Server," <http://www.redhat.com/docs/manuals/tux/>, (2001).
- [RND 1] RND Networks, "Web Server Director for Distributed Sites

- (WSD-DS),” *RND Networks Technical Applicaton Note 1035* - <http://www.rndnetworks.com/archive/pdfs/whitepapers/app1035.pdf>, (2001).
- [Schi01] B. N. Schilit, J. Trevor, D. M. Hilbert, and T. K. Koh, “m-Links: An Infrastructure for Very Small Internet Devices,” *Proc. 7th Ann. Int’l Conf. Mobile Computing and Networking (MobiCom 01)*, pp. 122-131 (July 2001).
- [Schn84] F. B. Schneider, “Byzantine Generals in Action: Implementing Fail-Stop Processors,” *ACM Transactions on Computer Systems* **2**(2) pp. 145-154 (May 1984).
- [Schn90] F. B. Schneider, “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,” *ACM Computing Surveys*, pp. 299-319 (December 1990).
- [Shen00] G. Shenoy, S. K. Satapati, and R. Bettati, “HydraNet-FT: Network Support for Dependable Services,” *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pp. 699-706 (April 2000).
- [Snoe00] A. C. Snoeren and H. Balakrishnan, “An End-to-End Approach to Host Mobility,” *Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiCom)*, pp. 155-166 (August 2000).
- [Snoe01] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, “Fine-Grained Failover Using Connection Migration,” *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pp.

- 221-232 (March 2001).
- [Ste99] L. Stein and D. MacEachern, *Writing Apache Modules with Perl and C*, O'Reilly and Associates (March 1999).
- [Sult01] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Highly Available Internet Services using Connection Migration," *Rutgers University Technical Report DCS-TR-462*, (November 2001).
- [Sury00] K. Suryanarayanan and K. Christensen, "Performance Evaluation of New Methods of Automatic Redirection for Load Balancing of Apache Web Servers Distributed in the Internet," *Proceedings of the IEEE 25th Conference on Local Computer Networks*, pp. 644-651 (November 2000).
- [Time02] TimesTen Inc, "Data Replication and TimesTen," <http://www.timesten.com>, (2002).
- [Toy78] W. N. Toy, "Fault-Tolerant Design of Local ESS Processors," *Proceedings IEEE* **66**(10) pp. 1126-1145 (October 1978).
- [Tren95] G. Trent and M. Sake, "WebSTONE: The First Generation in HTTP Server Benchmarking," <http://www.sgi.com/Products/WebFORCE/WebStone/paper.html>, (February 1995).
- [Wess99] D. Wessels, "Squid Web Proxy Cache," <http://www.squid-cache.org/>, (1999).
- [Zago03] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud,

- “Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP,”
*International Conference on Dependable Systems and Networks,
Dependable Computing and Communications Symposium (DSN-
2003)*, pp. 393 - 402 (June 2003).
- [Zand02] V. C. Zandy and B. P. Miller, “Reliable Network Connections,”
Proceedings of ACM MobiCom 2002, pp. 95-106 (September
2002).
- [Zhan04] R. Zhang, T. Abdelzaher, and J. Stankovic, “Efficient TCP
Connection Failover in Server Clusters,” *Proceedings of IEEE
INFOCOM*, pp. 1219-1228 (March 2004).
- [Zhao01] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, “Increasing the
Reliability of Three-Tier Applications,” *Proceedings of the 12th
International Symposium on Software Reliability Engineering*, pp.
138-147 (November 2001).