Computer Science Department Technical Report #TR050039
University of California, Los Angeles, September 2005.

CoRAL: A Transparent Fault-Tolerant Web Service

Navid Aghdaie and Yuval Tamir
Concurrent Systems Laboratory
UCLA Computer Science Department
Los Angeles, California 90095
{navid,tamir}@cs.ucla.edu

Abstract

The Web is increasingly used for critical applications and services. This paper presents a mechanism, called *CoRAL*, that provides high reliability and availability for Web service. CoRAL is client-transparent and provides fault tolerance even for requests being processed at the time of server failure. CoRAL does not require deterministic servers and can thus handle dynamic content. The scheme is based on a hot standby backup server that actively replicates the TCP connection state and maintains logs of HTTP requests and replies. For dynamic content, the throughput of a server cluster is increased by distributing the primary and backup tasks among the servers. For static content, that is deterministic and readily generated, the overhead is reduced by avoiding explicit logging of replies to the backup. In the event of a primary server failure, active client connections fail over to a spare, where their processing continues seamlessly. Our implementation of CoRAL is based on a Linux kernel module and an Apache Web server module. We describe key aspects of the design and implementation of CoRAL as well as several performance optimizations. We present measurements of system overhead during normal operation as well as failover performance and preliminary validation using fault injection.

1. Introduction

Many Internet services are critical to users and, as a revenue stream, to service providers. Erroneous processing or outages in applications such as online banking, stock trading, and shopping, are unacceptable. One of the key causes of service disruptions is server failures. Hence, fault tolerance techniques that allow providers to deliver continuous uninterrupted service despite server failures are increasingly important. For many services, such as Web service, there is a large installed base of clients and it is not practical to require all of those to be modified. Hence the fault tolerance scheme used should be *client-transparent*, i.e., operate without requiring any special action by the client. This transparency is a critical requirement with respect to both the client application and the client OS (e.g., TCP implementation).

Web service is often implemented as shown in Figure 1, using a three-tier architecture, consisting of: a client Web browser, one or more front-end servers (e.g. a Web server), and one or more back-end servers (e.g. a database). The front-end Web server is responsible for receiving the client requests and replying with a status code and either the content of a file, return value from a program/script which it executes, or results from a back-end server with which it communicates. The back-end server is often a database or application server that does the bulk of the required processing for each client request. The communication between the clients and the front-end servers use the HTTP protocol, which is typically implemented on top of TCP/IP.

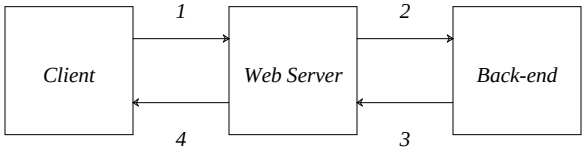

Figure 1: If the Web server fails before sending the client reply (step 4), the client can not determine whether the failure was before or after the Web server communication with the back-end (steps 2,3)

Web servers are often considered *stateless* since they do not maintain any state information from one client request to the next. Most of the relevant state is maintained by the back-end servers with the rest maintained by the client, as an HTTP cookie [30] or embedded in a URL. Hence, most existing Web server fault tolerance schemes simply detect failures and route *future* requests to backup servers that can start processing new requests in place of a failed server. Examples of such fault tolerance techniques include the use of specialized routers and load balancers [8, 9, 17, 19] and data replication [10, 46]. These methods are unable to recover requests being processed at failure time, henceforth referred to as *in-progress* requests.

While the Web server is stateless *between* transactions, it does maintain important state from the arrival of the first packet of a request to the transmission of the last packet of the corresponding reply. With the schemes mentioned above, the requests that were being processed are effectively dropped. Hence, depending on failure time, the clients may or may not receive complete replies to the in-progress requests and have no way to determine whether or not a

requested operation has been performed [1, 22, 23] (see Figure 1). A user of a standard Web browser client that fails to receive a reply may re-issue the request to be assured of its execution. However, the re-issuing of the request can lead to further problems since the same request may then be executed multiple times, possibly leading to undesired results. Seamless recovery of in-progress requests after a server failure requires a mechanism that recovers the state of the Web server — the state of active connections and generated replies. The key challenge is to do this in a way that does not require changes to the clients (client-transparency), that can handle non-deterministic servers that generate part of their content dynamically, and that has tolerable overhead.

In this paper we present CoRAL [1, 2, 4, 3], a fault-tolerant Web service scheme based on **Co**nnection **R**eplication and **A**pplication-level **L**ogging. Unlike most other schemes 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 addition to the basic design of the CoRAL scheme, the contributions of this paper include: 1) efficient implementation of CoRAL in a practical server environment using a combination of Linux kernel modules and Apache Web server modules, 2) performance optimizations of the basic scheme using load balancing and differential handling of static data, 3) evaluation of the performance overhead during normal operation as well as of the failover procedure, and 4) validation of the fault tolerance features using software fault injection.

The rest of this paper is organized as follows. In Section 2 we present the system architecture and key design choices. Section 3 is an overview of the assumptions underlying our work. Section 4 is a high level description of how our recovery mechanisms function. Section 5 presents our implementation based on Linux and Apache Web server modules. Performance evaluation and optimization are presented in Section 6. The results of software fault injection are presented in Section 7. Related work is discussed in Section 8.

2. CoRAL 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.

To achieve the fault tolerance goals, active replication of the servers may be used, where every client request is processed by two (or more) server replicas [15, 33, 40, 47]. During fault-free operation, the reply from only one of the servers is sent to the client. This approach suffers from several drawbacks. First, it has a high cost in terms of processing power, as every client request is effectively processed twice. Second, either servers are required to be deterministic [33, 40], or frequent synchronization between servers is required [15, 47]. Otherwise, the backup may not have an identical copy of a reply and thus it can not always continue the transmission of a reply should the primary fail in the midst of sending a reply.

An alternative approach is based on logging. Specifically, request packets are acknowledged only after they are stored redundantly (logged) so that they can be obtained even after a failure of a server host [1, 2, 7]. Since the server may be non-deterministic, none of the packets of a reply can be sent to the client unless the entire reply is safely stored (logged) so that its transmission can proceed despite a failure of a server host [1, 2]. The logging of requests can be done at the level of TCP packets [7] or at the level of HTTP requests [1, 2]. If request logging is done at the level of HTTP requests, the requests can be matched with logged replies so that a request will never be reprocessed following failure if the reply has already been logged [1, 2]. This is critical in order to ensure that for each request only one reply will reach the client. If request logging is done strictly at the level of TCP packets [7], it is possible for a request to be replayed to a spare server following failure despite the fact that a reply has already been sent to the client. Since the spare server may generate a *different* reply, two different replies for the same request may reach the client, violating the requirement for transparent fault tolerance.

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 [12] 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.

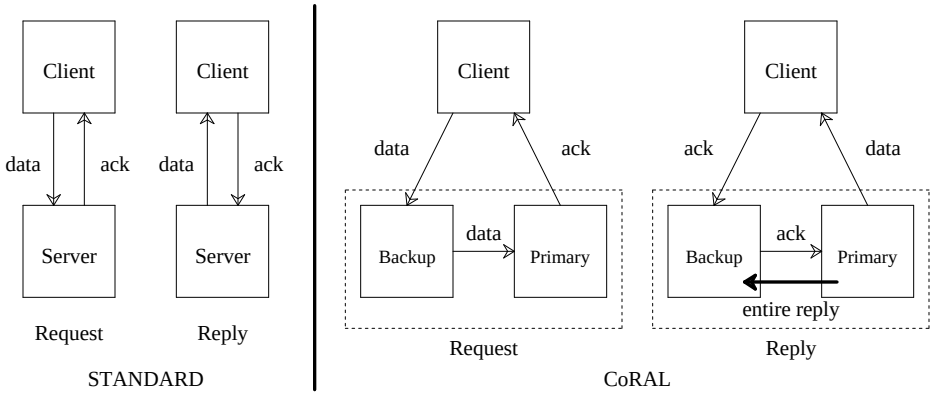

Figure 2: Message paths for a standard unreplicated server and CoRAL in fault-free (duplex) operation. CoRAL servers appear as a single entity to clients. The reply is generated by the primary and reliably sent to the backup before being sent to the client.

Figure 2 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 (Figure 1), 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 [36].

3. Assumptions

Fault injection experiments on other systems [32] have shown that most transient hardware faults either have no effect, cause a process to crash or cause the entire node to crash. 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 [49].

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/hang the entire host, i.e., *fail-stop* hosts [39], or to crash a process. As discussed later in Subsection 5.3, 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 [28]. 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. Recovery from Server Host Failures and Packet Loss

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. 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 1 summarizes the mechanisms used to recover from server failures that occur during different phases of handling HTTP requests and replies.

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 1: Recovery 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

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 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.

by TCP's error handling mechanisms. Table 2 summarizes possible communication errors and the mechanisms used to recover from them.

5. CoRAL Implementation

The basic scheme described in Section 2 can be implemented in different ways. Our implementation using user-level proxies [1] does not require any kernel modifications, and requires minimal (if any) changes to the Web server software. Although the user-level proxy implementation is simpler and potentially more portable, its associated performance overheads proved too costly for practical settings. In particular, we have shown elsewhere [2] 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. Hence, in this paper we focus on an approach based on a combination of kernel and Web server modifications [2].

The tasks of the scheme described in Section 2 can be divided into two categories: packet header modifications and operations performed at the HTTP message granularity. In our implementation, each of these categories of tasks is implemented by a separate module (Figure 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.

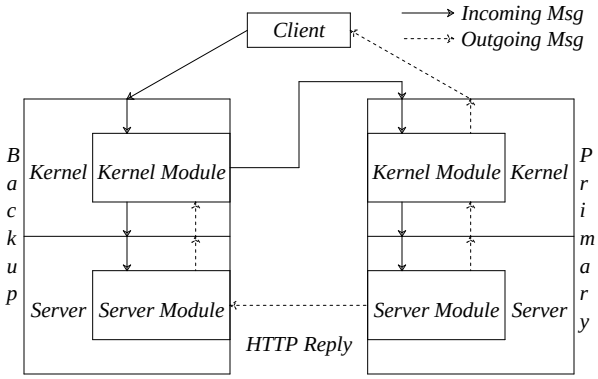

Figure 3: Implementation: kernel and Web server modules are used to provide the necessary mechanisms for replication. Message paths are shown.

The kernel module in our implementation is approximately 5000 lines of code. In addition, about 100 lines of code were added to the kernel itself, mainly for ‘‘hooks’’ that invoke functions in the kernel module. Almost all of this code is usable for other network services that use TCP connections [5]. The Apache server module is roughly 6500 lines of code. Most of this code would be usable for other request-reply services. The Apache code itself (approximately 100,000 lines of code) was not modified.

5.1. Kernel Modifications

The kernel modifications implement the client-transparent atomic multicast mechanism between the client and the primary/backup server pair. In addition, with these modifications the transmission of outgoing messages (replies) from the server pair to the client can continue seamlessly despite the failure of either server replica. The kernel modifications change the operation of the kernel as follows: 1) a copy of each incoming client packet arriving at the backup is 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 consistent sequence numbers.

In normal operation, the advertised address of the service known to clients is mapped to the backup server. Hence, the backup will receive all packets from the clients. After an incoming packet goes through the standard kernel operations such as checksum checking, the backup's kernel module 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. Such packets must be presented to the client with the advertised address as the source address. Hence, the primary's kernel module changes 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 kernel module intercepts and drops the packet when it reaches the device queue. TCP acknowledgments for outgoing packets to the client are, of course, incoming packets from the client and they are multicast to the primary and backup as above.

Another issue that the kernel module addresses 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 implementations select initial sequence numbers based on a random number. 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 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 kernel module at the primary server expects this value and uses it as it's initial sequence number. A possible alternative approach is to wait for the client to acknowledge a SYN packet sent by the primary, and synchronizing the backup initial server sequence number by using the acknowledgement number sent by the client [33].

5.2. Web Server Modifications

The server modifications, implemented as an Apache module, handle the parts of the scheme that deal with messages at the HTTP level. The operations of the Web server are modified to: 1) log arriving requests at the backup, 2) implement user-level reliable reply logging where each generated reply on the primary is logged to the backup, and 3) garbage collect the logged requests on the backup when the matching reply arrives from the client. Our Apache module acts as a handler [42] and generates the replies that are sent to the clients. It is composed of worker, mux, and demux processes described below.

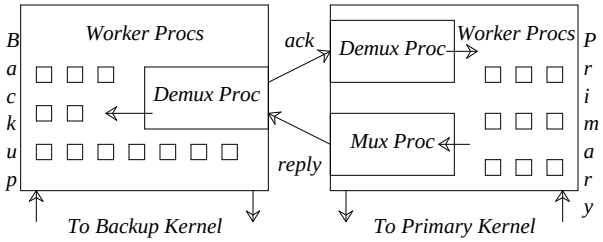

Figure 4: Server Structure: The mux/demux processes are used to reliably transmit a copy of the replies to the backup before they are sent to clients. The server module implements these processes and the necessary changes to the standard worker processes.

5.2.1. 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.

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.

5.2.2. 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 direct communication between the two worker processes. However, many practical servers, such as Apache, employ multiple worker processes or threads. Hence, a mechanism is required to link the worker processes on the two replicas that are handling the same request. With Apache, the handoff of new connections to worker processes is done by the kernel (via *listen* and *accept* calls) and is not deterministic. Hence, for each new connection, a worker process may need a connection to a different worker process on the other member of the primary/backup host pair. The implementation challenge is for a low-overhead mechanism for setting up the correct primary-backup logging connection for each new client connection.

Our solution to the above problem is to add *mux* and *demux* processes to the primary and add a *demux* process to the backup (Figure 4). 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).

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 user-level acknowledgment back to the primary for each reply. The reply's connection identifier is included in the acknowledgment message. The backup demux 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.

5.3. Converting Process Crashes to Host Crashes

As discussed in Section 3, 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 *or* crash or hang 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 crashed 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 distinguishing 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 [5]. 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 5.2, 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 5.4). 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 failed host (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.

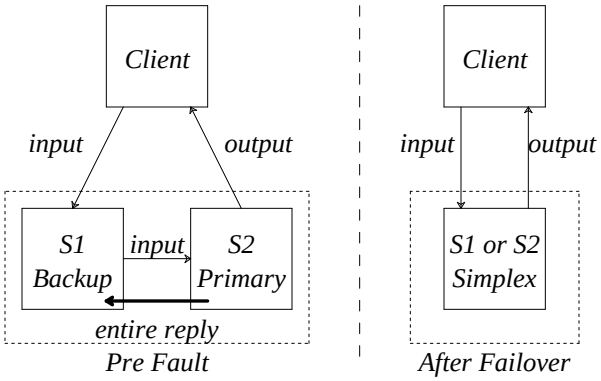

Figure 5: In normal (pre-fault) operation, client input is sent to the backup and then forwarded to the primary. The primary generates the output, logs it to the backup, and sends it to the client. After a fault, the surviving node takes over the identity of the failed node and operates in simplex mode.

5.4. Service Failover

As described above, for error detection, heartbeat messages are exchanged between server hosts. A user-level process on each server periodically sends sequenced UDP packets to its counterpart. Consecutive missed heartbeats signal that a failed host has been detected.

When a failed host is detected, the system must transition from standard replicated (duplex) operation to single server (simplex) mode (Figure 5) [4]. 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.

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. In our implementation, 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 [38] 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 errors 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 an error. 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 an error. 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 the time a host fails, some of the server module processes on the other host 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 an error occurs, these waiting processes must be notified to no longer wait, since the events they are waiting for will not occur in post-error (simplex) operation. The process that detects the error 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 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.

After recovery from an error, 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. 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 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. Once all the required processes are initialized, the active simplex server invokes a system call we have added, notifying the kernel module to also transition to duplex mode. Packets for new connections received after the transition are processed in duplex mode.

When the duplex configuration is restored, our implementation does not transition the existing client connections being processed in simplex mode to duplex mode. Hence, the kernel module and Web server module support a mode where simplex and duplex connections are

active simultaneously [4]. As an optimization, in order to ensure that simplex connections are short-lived, we use a feature of the standard HTTP/1.1 protocol [21] that allows the server to cause the client to cleanly close and re-establish the connection. New connections are processed in duplex mode.

6. Performance Evaluation and Optimization

To evaluate the performance of the scheme, we consider the overhead during normal operation as well as the duration of any impact on the service when an error occurs. The overhead during normal operation can be expressed in terms of increased response time, an increase in the number of server CPU cycles consumed 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: $Response\_Time = \alpha \times Reply\_Length + \beta + Reply\_Generation$ (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_Length* 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.

In the rest of this section we present results of experimental evaluation of our implementation. Most of the experiments were performed on 350 MHz Intel Pentium II PC's 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. Our kernel and server modules were installed for the applicable experiments.

Server replies were generated dynamically using WebStone 2.0 benchmark's CGI workload generator [35, 45]. This benchmark calls a random function to generate each reply byte. With this reply generator, where γ is the cost per byte of generating the reply, Equation 1 becomes: $Response\_Time = \alpha \times Reply\_Length + \beta + \gamma \times Reply\_Length$ (Eqn 2). We used custom clients similar to those of the Wisconsin Proxy Benchmark [6]. The clients continuously generate one outstanding HTTP request at a time with no think time. The request message length was approximately 50 bytes. For each experiment, the requests were for replies of a fixed size. Our measurements included reply sizes in the range of a few byte to 50K bytes (Internet traffic studies [13, 18] 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.

6.1. Latency Overhead

Figure 6 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

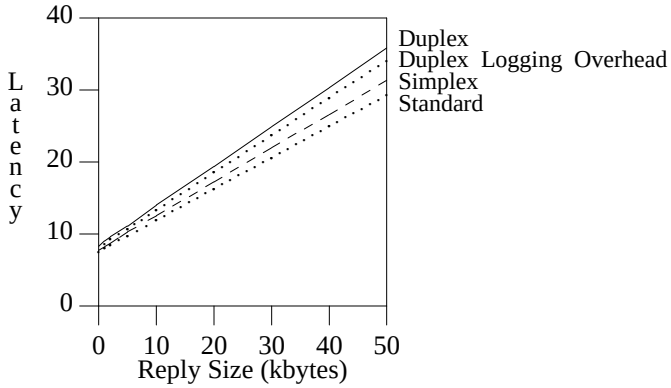

Figure 6: Average latency (msec) observed by a client for different reply sizes and system modes.

Equation 2, with $\gamma = 0.34$ μs/B, and for (standard, simplex, duplex), respectively, $\alpha = (0.10, 0.14, 0.21)$ μs/B, $\beta = (5.59, 5.86, 6.40)$ 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 6, 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 [34], 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.

6.2. Processing Overhead

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 [37] which utilizes the processor’s performance monitoring counter registers. Specifically, we measured global_power_events [26] 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.

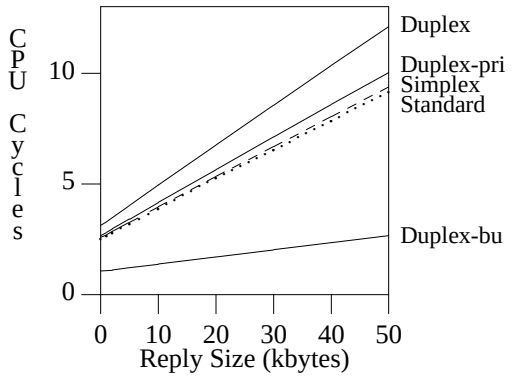

Figure 7: Used CPU cycles (in million) by different system modes for processing requests for different reply sizes. The primary and backup nodes of the system in duplex mode are depicted by *Duplex-pri* and *Duplex-bu* respectively. The Duplex line is the summation of primary and backup results.

Figure 7 shows the measurements of our implementation. We measured the CPU cycles used to receive one request and generate a reply. As mentioned earlier, these results can be modeled by an equation that has the same form as Equation 2:

$Cycles\_Used = \omega \times Reply\_Length + \psi + \phi \times Reply\_Length$ (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 transmits (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 7, 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 [2] 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 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.

If a server does have static, deterministic content, it is possible to significantly reduce the overhead using a modification of our scheme [3]. 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. Our measurements show that with this optimization, 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. With Equation 3, this yields parameter values of: $\phi = 0$, $\omega = 31$, $\psi = 0.778$. Hence, 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%. If the server has both static and dynamic content, the content type can be differentiated based on a component of the URL [3] so that the logging of replies is only done for dynamic content.

6.3. Throughput Overhead

With the experimental setup described in the beginning of this section, for dynamically generate replies, the maximum throughput is limited by the number of CPU cycles required per request. Figure 8 presents the measured peak throughput for different system configurations. As shown in Figure 7, 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*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.

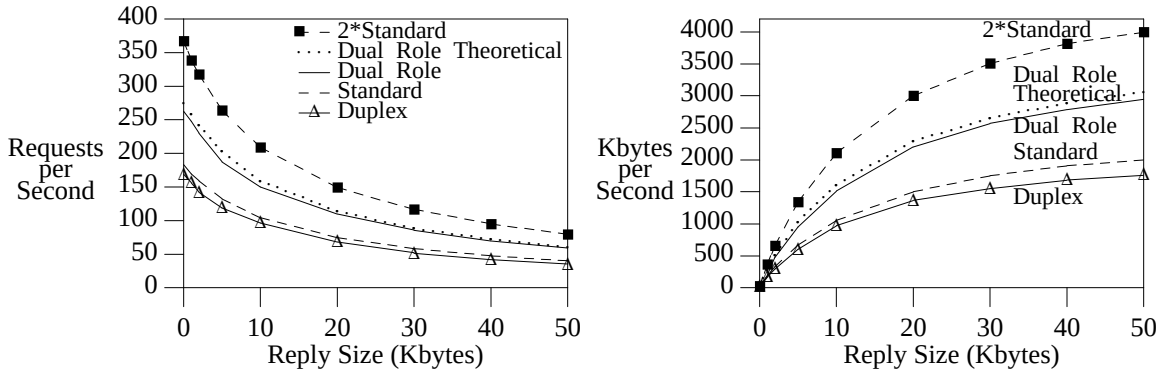

Figure 8: Peak system throughput for different reply message sizes (kbytes). Replies generated with WebStone 2.0 CGI benchmark.

The large throughput overhead of the *duplex* configuration is due to the fact that only the primary processes the requests. The backup does not execute the application, and during fault-free operation, it only performs our connection replication and message logging. Hence, the backup is mostly idle and its processing potential is largely wasted, especially for processor-intensive applications, such as serving dynamic content [3]. A simple solution to this problem is to distribute the primary server tasks and backup server tasks among both hosts. Hence, each server host serves simultaneously as the primary for some requests and as the backup for others. We refer to our implementation of this scheme as *dual-role servers* [3].

The *dual role* results show significant throughput improvement over duplex results (Figure 8) — 70%-72% of the *Standard* throughput (on two hosts) is achieved. 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.

6.4. Service Failover

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 [4]. 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 requests for 1 Kbyte dynamically-generated replies.

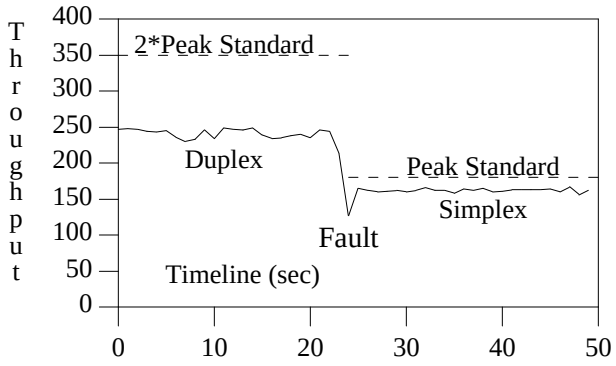

Figure 9: System throughput (in requests per second) for 1 Kbyte HTTP replies before and after a fault. Dashed lines indicate the upper bound, i.e., the peak throughputs of one and two unreplicated servers serving the same content.

Figure 9 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.

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 error detection and failover from duplex to simplex mode. Our measured failover transition time from fault detection in duplex mode to system execution in simplex mode is just 1.5 milliseconds (see Table 3). Since the transition time is short, most of the overall failover time is the time for error detection. The heartbeat period determines the error detection time. In our experiments we used a heartbeat period of 100 ms and a fault was assumed when two consecutive heartbeats were missed. As a result, the unavailability period was between 100 and 200 ms.

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

Table 3: 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.

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.

6.5. 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 6.3) generating dynamic 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.

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 5.2.2) is not scheduled frequently and the kernel receive buffer for the logging connection may become full. 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.

6.6. 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 2.66 GHz 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. The connections to the clients were always through a 100Mb/sec switched ethernet.

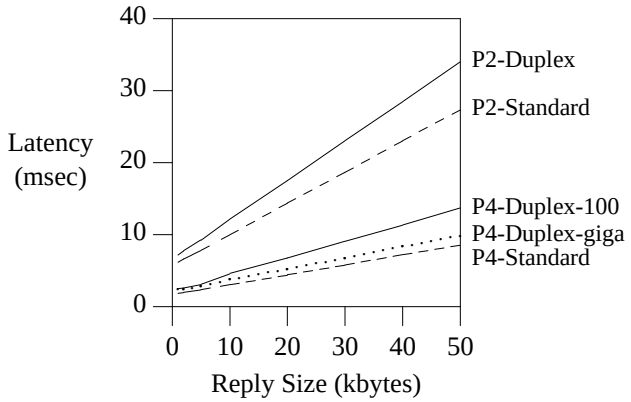

Figure 10: Average latency (msec) observed by a client for Pentium II (P2) and Pentium 4 (P4) server hosts. P4-Duplex-100 and P4-Duplex-giga denote P4 system in duplex mode with 100 Mbit/sec and gigabit/sec network configuration respectively.

Figure 10 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 $\gamma = 0.0518\ \mu s/B$, and for (standard, duplex), respectively, $\alpha = (0.0856, 0.102)\ \mu s/B$,

β = (1.67, 2.09) 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.

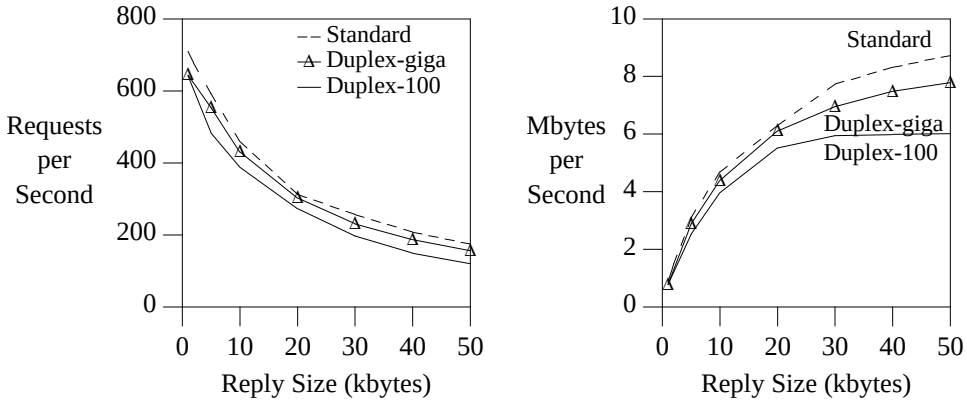

Figure 11: Peak system throughput for Pentium IV server hosts. Duplex-100 and Duplex-giga denote system in duplex mode with 100 Mbit/sec and gigabit/sec network configuration respectively.

With the P2 servers, the duplex configuration reached a peak throughput of 1.8MB/sec (Figure 8). As shown in Figure 11, 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 using a faster inter-server connection.

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 [24]. 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 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. We conducted targeted fault injection experiments, flipping bits in registers, or specific areas of memory. 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 from failures and inspected the replies received by clients for corruption. After a fault that caused a server host to crash (Subsection 5.3), the system would restore itself to duplex operation (Subsection 5.4), thus allowing the experiments to run mostly unattended. 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 [32].

Register fault injection was done by randomly selecting a register and then randomly selecting one bit of that register and inverting its value. This was done at a rate of one fault every five seconds. 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.

Our implementation uses large (30MB) shared memory segment for communication between web server and CoRAL processes. As a result, injections into the entire memory space resulted in only 1 corrupt reply and no crashes or hangs in over 70000 injections, at a rate of 2 faults/sec. As a second ‘‘stress test,’’ we excluded from injection 98% of the memory that included the shared memory segment and other large dynamically-allocated segments that are sparsely used. In this case, 10,358 injections resulted in 1 corrupted reply, 3 hangs, and 84

crashes. Full recovery was achieved from all the crashes. As a third test, the injection was restricted to the stack segment. In this case, 10114 injections resulted in no corrupted replies, 5 hangs, and 211 crashes. Full recovery was achieved from all the crashes.

8. Related Work

There are many schemes for increasing the *availability* of network services by providing a mechanism to route new requests that arrive after a fault to a working server. Such schemes do not support recovery of in-progress requests. Approaches include the use of DNS system to remove the address of faulty servers from service [14], centralized router schemes [8, 16, 17, 27] which route client packets only to working servers, and redirect schemes [44] which direct clients to an alternate server replica.

There are various fault-tolerance schemes for network services that are *not* client transparent. Some require modifications to the client application while others only require changes to the client OS kernel (the TCP implementation). Implementations of a two-phase commit protocol on a three-tier system [23, 50] ensure that transactions are performed exactly once and that in-progress requests are recovered. There are various techniques that, with participation of the client application and/or OS, allow active connections can be migrated from a faulty server to a backup replica [41, 43, 48]. Since client applications, such as Web browsers, and client operating systems are not under the control of service providers, relying on client-side participation for fault tolerance is currently not practical. Recently proposed client-aware reliability standards [11, 20, 25], can achieve fault tolerance with a lower overhead. In the future, if these standards are widely adopted, such client-aware solutions may become preferable.

Over the last five years there have been several research projects focused on client-transparent fault tolerance for Web service. HydraNet-FT [40] uses actively replicated servers. Specialized routers (‘‘redirectors’’) are used to multicast client packets and each request is processed on all replicas. The redirectors are a single point of failure. Also, since HydraNet-FT uses only active replication, it requires deterministic servers while CoRAL does not have this

restriction. However, while CoRAL requires the servers to be co-located on the same subnet, HydraNet-FT has the advantage of allowing the server replicas to be geographically distributed.

FT-TCP [7, 47] uses the log and replay approach. It is implemented as a kernel-level TCP wrapper that transparently masks server failures from clients. In the original implementation [7], all incoming client packets are logged on a logger running on a separate processor. To recover from an error, all logged packets are replayed on an initialized server. It is assumed that the server application behavior is mostly deterministic (with a few exceptions[7]). Hence the replaying of logged messages will lead to the recovery of pre-fault server state. A protocol similar to that of our kernel module is used to achieve client transparency. An improved version of the scheme [47] uses a backup host instead of the logger, handles non-determinism, and allows for system operation in either cold (logging) or hot (active replication) modes. Non-deterministic execution is handled by synchronizing the system call results of the primary and backup hosts. Hotswap [15] also uses this approach to handle some level of non-determinism. FT-TCP was used and evaluated for implementing a fault-tolerant streaming server and a file/printing server. It has not been used with a Web server. No fault injection results were presented for FT-TCP.

Some implementations [29, 31, 33, 49] assume the existence of a shared network at the server site and thus avoid some of the overhead and complexity associated with the transparent multicast of client packets (e.g., CoRAL kernel module). ST-TCP [33] uses a backup which snoops client packets that are intended for a primary server. The primary does not discard packets until the backup acknowledges their receipt. In addition, a separate logger node, which logs all incoming packets, is required to handle primary failures along with a dropped packet at the backup. Transparent TCP Connection Failover [29] eliminates the need for a logger by forwarding the backup's outgoing (acknowledgement) packets to the primary, and delaying the primary's outgoing packets until a matching packet is received from the backup. Luo and Yang [31] also rely on snooping in their implementation by using IP aliasing. They do not use a packet logger, hence there is a possibility that a client packet may be acknowledged even though

it has only been received by one of the servers. Thus, the states of the primary and backup may diverge, possibly leading to an inability to recover from some failures.

An initial, user-level, implementation of some of the basic ideas of CoRAL [1] resulted in high overhead, motivating the current approach of relying on kernel-level support. Preliminary descriptions of various aspects of CoRAL have been presented previously [2, 3, 4]. However, in this paper the analysis of performance evaluation experiments is redone and there is new material on the mechanism for converting process crashes to host crashes, handling of the scheme under heavy load, the evaluation of the impact of server host speed, and the validation using fault injection.

9. Conclusion

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.

References

1. N. Aghdaie and Y. Tamir, ‘‘Client-Transparent Fault-Tolerant Web Service,’’ *20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, AZ, pp. 209-216 (April 2001).

2. N. Aghdaie and Y. Tamir, ‘‘Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support,’’ *11th IEEE International Conference on Computer Communications and Networks*, Miami, FL, pp. 63-68 (October 2002).

3. N. Aghdaie and Y. Tamir, ‘‘Performance Optimizations for Transparent Fault-Tolerant Web Service,’’ *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada, pp. 29-32 (August 2003).

4. N. Aghdaie and Y. Tamir, ‘‘Fast Transparent Failover for Reliable Web Service,’’ *International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, pp. 757-762 (November 2003).

5. N. Aghdaie and Y. Tamir, ‘‘Efficient Client-Transparent Fault Tolerance for Video Conferencing,’’ *International Conference on Communication and Computer Networks*, Marina del Rey, CA (October 2005).

6. J. Almeida and P. Cao, ‘‘Wisconsin Proxy Benchmark,’’ *Technical Report 1373, Computer Sciences Dept, Univ. of Wisconsin-Madison* (April 1998).

7. L. Alvisi et al., ‘‘Wrapping Server-Side TCP to Mask Connection Failures,’’ *IEEE INFOCOM*, Anchorage, AK, pp. 329-337 (April 2001).

8. E. Anderson et al., ‘‘The Magicrouter, an Application of Fast Packet Interposing,’’ *Class Report, UC Berkeley - http://www.cs.berkeley.edu/~eanders/projects/magicrouter/* (May 1996).

9. D. Andresen et al., ‘‘SWEB: Towards a Scalable World Wide Web Server on Multicomputers,’’ *Proccedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, pp. 850-856 (April 1996).

10. S. M. Baker and B. Moon, ‘‘Distributed Cooperative Web Servers,’’ *The Eighth International World Wide Web Conference*, Toronto, Canada, pp. 1215-1229 (May 1999).

11. R. Bilorusets et al., *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*, March 2003.

12. A. Borg et al., ‘‘A Message System Supporting Fault Tolerance,’’ *9th Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).

13. L. Breslau et al., ‘‘Web Caching and Zipf-like Distributions: Evidence and Implications,’’ *IEEE INFOCOM*, New York, New York (March 1999).

14. T. Brisco, ‘‘DNS Support for Load Balancing,’’ RFC 1794, IETF (April 1995).

15. N. Burton-Krahn, ‘‘HotSwap - Transparent Server Failover for Linux,’’ *USENIX LISA ’02: Sixteenth Systems Administration Conference*, Berkeley, California, pp. 205-212 (November 2002).

16. 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*.

17. 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*.

18. C. Cunha et al., ‘‘Characteristics of World Wide Web Client-based Traces,’’ *Technical Report TR-95-010, Boston University, CS Dept, Boston, MA 02215* (April 1995).

19. D. M. Dias et al., ‘‘A scalable and highly available web server,’’ *IEEE COMPCON ’96*, San Jose, California, pp. 85-92 (1996).

20. C. Evans et al., ‘‘Web Services Reliability (WS-Reliability),’’ http://www.oasis-open.org/committees/wsrm/charter.php, OASIS Web Services Reliable Messaging Technical Committee, Working Draft (January 2003).

21. R. Fielding et al., ‘‘Hypertext Transfer Protocol -- HTTP/1.1,’’ RFC 2616, IETF (June 1999).

22. S. Frolund and R. Guerraoui, ‘‘CORBA Fault-Tolerance: why it does not add up,’’ *IEEE Workshop on Future Trends of Distributed Systems* (December 1999).

23. S. Frolund and R. Guerraoui, ‘‘Implementing e-Transactions with Asynchronous Replication,’’ *IEEE International Conference on Dependable Systems and Networks*, New York, New York, pp. 449-458 (June 2000).

24. M.-C. Hsueh et al., ‘‘Fault Injection Techniques and Tools,’’ *IEEE Computer* **30**(4), pp. 75-82 (April 1997).

25. IBM Inc, ‘‘HTTPR Specifications,’’ *http://www-106.ibm.com/developerworks/webservices/library/ws-httprspec/* (April 2002).

26. Intel Inc, *IA-32 Intel Architecture Software Developer’s Manual - Volume 3: System Programming Guide*, 2004.

27. Intel Inc, ‘‘Intel NetStructure e-Commerce Products,’’ *http://www.intel.com/support/netstructure/commerce/index.htm*.

28. S. Knight et al., ‘‘Virtual Router Redundancy Protocol,’’ RFC 2338, IETF (April 1998).

29. R. Koch et al., ‘‘Transparent TCP Connection Failover,’’ *International Conference on Dependable Systems and Networks (DSN-2003)*, San Francisco, California, pp. 383-392 (June 2003).

30. D. Kristol and L. Montulli, ‘‘HTTP State Management Mechanism,’’ RFC 2109, IETF (February 1997).

31. M.-Y. Luo and C.-S. Yang, ‘‘Constructing Zero-Loss Web Services,’’ *IEEE INFOCOM*, Anchorage, AK, pp. 1781-1790 (April 2001).

32. H. Madeira et al., ‘‘Experimental Evaluation of a COTS System for Space Applications,’’ *International Conference on Dependable Systems and Networks*, Washington, D.C., pp. 325-330 (June 2002).

33. M. Marwah et al., ‘‘TCP Server Fault Tolerance Using Connection Migration to a Backup Server,’’ *International Conference on Dependable Systems and Networks (DSN-2003)*, San Francisco, California, pp. 373-382 (June 2003).

34. Matrix NetSystems Inc, ‘‘The Internet Weather Report,’’ *http://www.internetweather.com*.

35. Mindcraft Inc, ‘‘WebStone Benchmark Information,’’ *http://www.mindcraft.com/webstone*.

36. Oracle Inc, *Oracle8i Distributed Database Systems - Release 8.1.5*, Oracle Documentation Library (1999).

37. M. Pettersson, ‘‘Linux x86 Performance-Monitoring Counters Driver,’’ *http://www.csd.uu.se/˜mikpe/linux/perfctr/*.

38. D. C. Plummer, ‘‘An Ethernet Address Resolution Protocol,’’ RFC 826, IETF (November 1982).

39. F. B. Schneider, ‘‘Byzantine Generals in Action: Implementing Fail-Stop Processors,’’ *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).

40. G. Shenoy et al., ‘‘HydraNet-FT: Network Support for Dependable Services,’’ *20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 699-706 (April 2000).

41. A. C. Snoeren et al., ‘‘Fine-Grained Failover Using Connection Migration,’’ *3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, pp. 221-232 (March 2001).

42. L. Stein and D. MacEachern, *Writing Apache Modules with Perl and C*, O’Reilly and Associates (March 1999).

43. F. Sultan et al., ‘‘Migratory TCP: Highly Available Internet Services using Connection Migration,’’ *Rutgers Univesity Technical Report DCS-TR-462* (November 2001).

44. K. Suryanarayanan and K. Christensen, ‘‘Performance Evaluation of New Methods of Automatic Redirection for Load Balancing of Apache Web Servers Distributed in the Internet,’’ *IEEE 25th Conference on Local Computer Networks*, Tampa, Florida, pp. 644-651 (November 2000).

45. G. Trent and M. Sake, ‘‘WebSTONE: The First Generation in HTTP Server Benchmarking,’’ *http://www.sgi.com/Products/WebFORCE/WebStone/paper.html* (February 1995).

46. R. Vingralek et al., ‘‘Web++: A System For Fast and Reliable Web Service,’’ *USENIX Annual Technical Conference*, Sydney, Australia, pp. 171-184 (June 1999).

47. D. Zagorodnov et al., ‘‘Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP,’’ *IEEE International Conference on Dependable Systems and Networks*, San Francisco, CA, pp. 393-402 (June 2003).

48. V. C. Zandy and B. P. Miller, ‘‘Reliable Network Connections,’’ *ACM MobiCom*, Atlanta, Georgia, pp. 95-106 (September 2002).

49. R. Zhang et al., ‘‘Efficient TCP Connection Failover in Server Clusters,’’ *IEEE INFOCOM*, Hong Kong, pp. 1219-1228 (March 2004).

50. W. Zhao et al., ‘‘Increasing the Reliability of Three-Tier Applications,’’ *12th International Symposium on Software Reliability Engineering*, Hong Kong, pp. 138-147 (November 2001).