Open access • Journal Article • DOI:10.1145/1534909.1534911

Practical and low-overhead masking of failures of TCP-based servers — Source link

Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, Thomas Bressoud

Institutions: University of California, Santa Barbara, University of California, San Diego, University of Texas at Austin, Denison University

Published on: 29 May 2009 - ACM Transactions on Computer Systems (ACM)

Topics: TCP acceleration, Server farm, Compound TCP, Server and Client–server model

Practical and Low-Overhead Masking of Failures of TCP-Based Servers

4

DMITRII ZAGORODNOV
University of California, Santa Barbara
KEITH MARZULLO
University of California, San Diego
LORENZO ALVISI
The University of Texas at Austin
and
THOMAS C. BRESSOUD
Denison University

This article describes an architecture that allows a replicated service to survive crashes without breaking its TCP connections. Our approach does not require modifications to the TCP protocol, to the operating system on the server, or to any of the software running on the clients. Furthermore, it runs on commodity hardware. We compare two implementations of this architecture (one based on primary/backup replication and another based on message logging) focusing on scalability, failover time, and application transparency. We evaluate three types of services: a file server, a Web server, and a multimedia streaming server. Our experiments suggest that the approach incurs low overhead on throughput, scales well as the number of clients increases, and allows recovery of the service in near-optimal time.

Categories and Subject Descriptors: D.4.4 [**Operating Systems**]: Communications Management—*Network communication*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.8 [**Operating Systems**]: Performance—*Measurements*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Fault-tolerant computing system, primary/backup approach, TCP/IP

Authors' addresses: D. Zagorodnov, Computer Science Department, University of California, Santa Barbara, Santa Barbara, CA 93106; K. Marzullo, Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive #0404, La Jolla, CA 92093-0404; L. Alvisi, Department of Computer Sciences, College of Natural Sciences, The University of Texas at Austin, 1 University Station C0500, Austin TX 78712; T. C. Bressoud, Department of Computer Science, Denison University, Granville, OH 43023.

DOI 10.1145/1534909.1534911 http://doi.acm.org/10.1145/1534909.1534911

ACM Reference Format:
Zagorodnov, D., Marzullo, K., Alvisi, L., and Bressoud, T. C. 2009. Practical and low-overhead masking of failures of TCP-based servers. ACM Trans. Comput. Syst. 27, 2, Article 4 (May 2009), 39 pages. DOI = 10.1145/1534909.1534911 http://doi.acm.org/10.1145/ 1534909.1534911

1. INTRODUCTION

TCP is the most popular transport-layer protocol in use today and a diverse set of network services has been built on top of it. It is used for short sessions such as HTTP connections, for longer sessions that involve large data transfers, and for continuous sessions like those used by the interdomain routing protocol BGP. As more people come to rely on network services, the issue of reliability has become pressing. To ensure reliable service, failures of the service endpoint must be tolerated.

Many companies marketing high-end server hardware—IBM, Sun, HP, Veritas, Integratus—offer fault-tolerant solutions for TCP-based servers. The solutions are usually built using a cluster of servers interconnected with a fast private network which is used for access to shared disks, for replica coordination, and for failure detection. When a server in the cluster fails, all ongoing connections to that server break. The failover mechanism ensures that if a client attempts to reopen a connection, then it will be directed to a healthy server. Although this client-assisted recovery is adequate for some services, it is often desirable to hide server failures from clients.

When the client base is large and diverse, the organization running the service may lack control over the client host configuration and the applications running on the host. This means that client applications often cannot be expected to assist in the failover of the service. Such is the situation with many Internet services, where servers and clients are written by different people and provisions for fault tolerance in the application-level protocol do not exist. Convincing one's business clients or partners to upgrade to a new protocol is often not an option.

Consider a popular file server, Samba [SMB 2005]: If the server fails, all transfers are aborted and the user must explicitly restart any outstanding transactions. Although the protocol allows aborted transfers to be restarted, many clients, such as Windows Explorer, choose not to do this, entailing retransmission of entire files. Similarly, typical media streaming clients, such as Apple's QuickTime Player, do not attempt to restart aborted sessions, which means users can miss sections of live broadcasts even if they restart the stream manually. Web browsers do not hide server errors or failures by design; a partial HTTP response is rendered either as an error or as an incomplete Web page, depending on how much data the browser received. Often a transient Web server failure can be solved by the user pressing the browser's "reload" button; however, a server failing in the middle of a financial transaction can leave the user unsure whether the transaction took place or not. These are all examples of user irritations that a service provider may want to avoid.

In this article we describe a system called *fault-tolerant* TCP *(*FT-TCP*)*. This system allows a faulty server to keep its TCP connections open until it either

recovers or is failed over to a backup. In either case, both the failure and recovery of the server are completely transparent to clients connected to it via TCP. FT-TCP does not require any changes to the client software, does not require changes to the TCP protocol, and does not use a proxy in the middle of the network; all fault-tolerance logic is constrained to the server cluster. Furthermore, because the system has been designed in the form of "wrappers" around kernel components, no changes to the TCP stack implementation on the server are required, while the required changes to server applications are small.

We have evaluated the performance of FT-TCP both with a synthetic application designed to obtain maximum throughput of TCP, as well as with several real-world services, such as Samba [SMB 2005], Darwin Streaming Server [DSS 2005], and the Apache [2005] Web server. FT-TCP supports two common application-level replication methods: *primary-backup* [Budhiraja et al. 1992] and *message-logging* [Elnozahy et al. 2002]. In our experiments, we found their failure-free performance statistically indistinguishable. Neither one incurred significant overhead on connection throughput for bulk transfers, while their effect on latency depends on the client request traffic: With many small requests from few clients, the overhead is large, but as either the request size or the number of clients grows, the overhead diminishes to the point of insignificance. We also found that with primary-backup the failover time of FT-TCP can be made short, but to do so the backup must aggressively capture client data.

The remainder of this article is organized as follows. We cover the background material relevant to this work in Section 2. We offer an overview of the general structure of our system (primary-backup as well as message-logging versions) in Section 3, while Sections 4–6 present FT-TCP in greater detail. In Section 7 we describe the three applications we use to evaluate the system. The performance discussion is divided in two parts: In Section 8 we discuss the overhead of FT-TCP in terms of throughput and latency, and in Section 9 we look at the dynamics of connection failover. We compare FT-TCP to other possible approaches and alternative systems in Section 10. Finally, we draw our conclusions in Section 11.

2. BACKGROUND

In this section, we introduce two concepts that are relevant to the discussion of service failover: operation of TCP and fault tolerance fundamentals.

2.1 TCP Overview

TCP implements a bidirectional byte stream by fragmenting data into segments and by sending each one in a packet with its own header. (The maximum size of the segment is limited to 40 bytes less than the Maximum Transmission Unit, or MTU, of the path between the sender and the receiver, which is usually 1500 bytes long, making the typical maximum segment size 1460 bytes.) The header carries control data, implementing error recovery, flow control, and congestion control.

To this end, the header carries two *sequence numbers*, one for each direction. When a connection is established, each connection endpoint selects a random

32-bit integer to serve as its *initial sequence number (isn)* that is logically associated with an imaginary byte 0 in the data stream. Consequently, an actual byte number n (where $n \geq 1$) in the stream is associated with the sequence number $(isn + n)\ mod\ 2^{32}$. The modulo operation accounts for the sequence number wraparound that occurs when the number exceeds the capacity of a 32-bit integer. Every header contains the sequence number of the first byte in the segment that the packet carries, allowing the receiver to sequence that segment relative to all other segments regardless of the order in which they arrive. Duplicates are likewise detected and ignored.

TCP connections are established with the help of binary flags in the packet header. A client initiates the connection by sending to the server a packet with the SYN flag set and with a randomly chosen sequence number isn_c. If the server *accepts* the connection (i.e., the server is willing and able to proceed with this client) it replies back with a packet that has both the SYN and ACK flags set and contains a proposed isn_s for the server as well as the TCP header *acknowledgment number* field, set to $isn_c + 1$. Outgoing acknowledgment numbers are set to the sequence number of the byte following the last contiguous byte the receiver got from the sender, thereby indicating what data have been received. We call a packet that acknowledges data but does not carry any data an ACK *packet*, or simply an ACK. Finally, the client replies with an ACK packet with acknowledgment number set to $isn_s + 1$, at which point both sides consider the connection established. This protocol is known as a *three-way handshake*. We call the byte stream from the client to the server the *instream* and the byte stream from the server to the client the *outstream*.

To implement flow control, the TCP header carries a 16-bit *window size* field, which indicates to the sender how much buffer space is available on the receiver. If, for example, the advertised window is 16KB, then the sender can send up to eleven 1460-byte segments before stalling in wait for an acknowledgment. The window size is used for flow control: If the receiver is not able to process the incoming segments fast enough, the window shrinks and may eventually reach zero, at which point the sender refrains from sending any more segments. As the receiver consumes the buffered segments, its buffers free up and the window increases in size, allowing the sender to resume sending data.

To implement a reliable stream, TCP must deal with dropped or corrupted packets. A checksum of the whole packet enables TCP to identify corrupted packets and discard them. The acknowledgment number tells the client when packets are dropped using a cumulative acknowledgment scheme. For example, in a situation where packets A, B, and C are sent and packet B gets dropped, the receiver will acknowledge only A even after it receives C. Eventually, a retransmission timer will expire on the sender, which will then resend B, thus filling the gap and causing the receiver to acknowledge all three packets by acknowledging packet C.

2.2 Replication Concepts

Recovery of a network service is possible when every connection is backed by some number of server replicas: a primary server and at least one backup.

Should the primary fail, all backups must have the information needed to take over the role of the primary as endpoint in its ongoing connections. A backup is said to be *promoted* when it is chosen to become the next primary. FT-TCP supports two approaches to coordinating replicas.

In the first approach, called *primary-backup* [Budhiraja et al. 1992], every replica performs the processing of client requests; when all replicas have completed processing, one of them (the primary) replies. If the primary fails, one of the backup replicas is promoted. In the second approach, called *message logging* [Elnozahy et al. 2002], only one replica is actively processing requests and all requests from the client are saved in a log that is guaranteed to survive failures. Just as in the first approach, the primary does not reply to the client until it is assured that all prior requests have been logged. If a failure occurs, another replica is started. This replica replays messages from the log to bring itself to the prefailure state of the primary, at which point the replica is promoted. If periodic checkpoints are taken, then only the messages that arrived since the most recent checkpoint need to be replayed.

In this article, we refer to these two approaches as *hot backup* and *cold backup*. In both approaches the primary waits before replying to a client until it is assured that the backup can be recovered to the primary's current state. This is commonly called the *output commit* problem [Elnozahy et al. 2002]. We henceforth refer to these forced waiting periods as *output commit stalls*. When a backup takes over, it does not know whether the primary failed before or after replying to the client (this is a fundamental limitation of any fault-tolerant system). Fortunately, TCP was designed to deal with duplicate packets, so when in doubt the backup can safely resend the reply.

For both hot and cold backups, the process execution paths of the primary and the backups must match. If they do not, then a backup may never reach the state of the primary and therefore will not be able to take over the connection. If, for example, a system call returns different values on the primary and a backup replica, the execution paths of these processes may diverge. To accommodate this possibility, in addition to capturing client requests, we also intercept system calls on all replicas, save the primary's system call results in a log, and return those values as the results of the corresponding system calls on the replicas. We discuss further how we deal with this and other sources of *nondeterminism* in Section 5.

3. ARCHITECTURE OVERVIEW

FT-TCP enables connection failover for Internet services that is transparent to the client and requires no changes to the TCP protocol or to the operating system running on the server. The key idea is that, by logging incoming network data as well as the information that the service process receives from the operating system, a replica of the service can be created on backup machines and, if necessary, substituted for the original service without breaking its TCP connections.

To avoid changes to the operating system, FT-TCP is implemented by "wrapping" the TCP/IP stack. By this, we mean that FT-TCP can intercept, modify, and discard packets between the TCP/IP stack and the network driver using a component

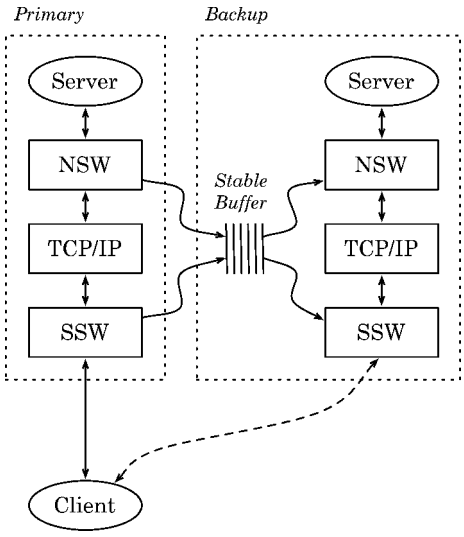

Fig. 1. FT-TCP architecture.

we call the *south-side wrapper* or SSW. Also, FT-TCP can intercept and change the semantics of system calls (between application and kernel) made by the server application using a component we call the *north-side wrapper* or NSW. Although the wrappers are kernel-level components, they require no changes to the operating system and, in our implementation, can be loaded into and unloaded from a standard kernel dynamically, without a recompilation or a reboot. Both the NSW and the SSW on the primary replica communicate with a *stable buffer* that is designed to survive failures.

In our implementation, the stable buffer is located in the physical memory of the backup machines, but other approaches, such as saving data on disk or in nonvolatile memory, are also possible. In addition to logging data, a stable buffer can acknowledge the data elements it receives, as well as return them to a requester in FIFO order. When we call a datum *stable*, we mean that it has been acknowledged by the stable buffer and will therefore survive a failure.

In the rest of the article we will use a setup with a single backup and a single stable buffer, colocated with that backup, as shown in Figure 1. Our technique can be extended to use any number of backup hosts by modifying the stable buffer protocol to use reliable broadcast and by ensuring that, during failover, all backups elect the same primary. Furthermore, in the discussion that follows, although we may talk of *one* server process or *one* connection, FT-TCP supports multiple concurrent processes and concurrent connections, possibly to a single process, and tolerates the failure of the primary as well as of any number of the replicas—as long as, of course, at least one backup replica continues to operate correctly.

3.1 Failure-Free Operation

FT-TCP begins interception either when a network service application that it is designated to protect begins execution or when the first client packet for that

service arrives from the network, whichever happens first. (The latter is the case when the service application is invoked by a metaserver, such as inetd.) During a run without failures, the SSW on the primary sends incoming packets to the stable buffer. The primary's NSW does the same with the results of system calls invoked by each thread of the server application. In the cold-backup scheme, the wraps on the backup replica are idle (they will be activated only if a failover is necessary), so the only activities performed by the backup machine are logging and monitoring of the health of the primary replica.

In the hot-backup scheme, in addition to logging and monitoring, the backup is executing the same server application binary as the one running on the primary. To this process the NSW "feeds" the contents of the stable buffer. Specifically, the NSW ensures that the backup process receives the same results from its system calls as does the primary process. In particular, this includes the recv call for receiving network input, which the NSW extracts from the packets in the stable buffer. The role of the SSW on the backup during failure-free operation is limited to spoofing incoming client connections so that the backup's TCP stack allocates state for them.

3.2 Failover

A failure of the primary is detected by the backup based on the absence of communication by the primary for an interval of time (see Section 9 for additional details on failure detection and recovery durations). Once the failure is detected, the backup initiates failover, which proceeds differently for the hot-backup and the cold-backup schemes.

In the hot-backup scheme, the backup replica already has a server process that has been following the execution path of the server process on the failed primary. Thus, after the backup process "catches up" to where the primary process was just before the failure (by consuming all remaining TCP segments and system call records in the stable buffer), it can take over the open connection. The failover completes with the backup machine being promoted to a primary. To enable the backup to impersonate the failed primary to the clients, the SSW acts to reconcile the externally visible differences in the TCP state between the old primary and the new one.

One such difference is the IP address. In our implementation, the SSW switches the backup's real IP address for the old primary's address on all outgoing packets and performs the reverse on all the incoming client packets, effectively functioning as a network address translation (NAT) unit [Srisuresh and Holdrege 1999]. To gain access to all incoming packets (that are destined to a different MAC address), we place the network interface card into promiscuous mode. When using a switched hub for connecting replicas to the client, the hub must be configured to direct client packets to all replicas. If some other technique for permanently changing the IP address of the entire host is used (e.g., by using a *gratuitous* ARP [Bhide et al. 1991] or a physically separate address translating switch), then using promiscuous mode may not be necessary.

The other difference in TCP connection state between the primary and the backup is in the sequence numbers they use. The TCP connection on the backup

is idle during normal operation (since all the data are injected through the NSW), so its sequence numbers stay at their initial values. After failover the sequence numbers must be adjusted by the SSW on all packets as follows: Incoming sequence numbers are shifted by the number of bytes the primary read before failure and the outgoing ones are adjusted by the difference between the backup's initial sequence number and the sequence number of the last byte the primary sent to the client.

Failover with a cold backup essentially consists of redoing the actions performed by a hot backup during normal operation followed by the actions performed by it during failover. First, a new server process is started on the backup host and a connection to it is spoofed by the SSW (in the case of a metaservice, such as inetd, the spoofing of the connection will cause the creation of a new server process). This process then "consumes" buffered packets and system calls, and eventually takes over the connection after the IP address and sequence number adjustments described earlier. Since rolling the process forward takes time, failover with a cold backup can take substantially longer than with a hot backup. Recovery from a cold backup can be sped up significantly by adding a checkpointing mechanism to FT-TCP; however, checkpointing the state of the server application is outside of the scope of this article, henceforth, we assume that a restarting server has the application restart from its initial state.

Failure of the backup is detected by the primary based on the absence of acknowledgments from the backup for an interval of time. Henceforth, the failed backup does not cause any further output commit stalls at the primary.

During failover, it is important to prevent connection timeouts and the appearance of a nonresponsive server. In FT-TCP, a separate component keeps client connections alive by responding to their segments with an ACK packet that has a window of size 0. This gives clients the illusion that the server is still viable, but also does not allow them to send any more data while the service is recovering.

4. ARCHITECTURE DETAILS

In this section, we describe the operation of FT-TCP in detail. After introducing the state maintained by FT-TCP, we describe the activities of the wrappers in different modes of operation.

4.1 State

FT-TCP maintains the following variables for each ongoing connection.

—*idelta_seq* and *odelta_seq*: the deltas (for instream and outstream) between the sequence numbers in use by the client and the sequence numbers apparent to the TCP stack at the server. Upon recovery, these variables allow the SSW to map sequence numbers between the server's TCP layer and the client's TCP layer for the instream and outstream.

—*stable_seq*: the smallest sequence number of the instream that has not yet been acknowledged by the stable buffer to the SSW. The value of *stable_seq* can never be larger (ignoring the 32-bit wrap) than the largest sequence number

Table I. Summary of the Key Variables Used by FT-TCP

Variable	Updated	Related to
idelta_seq	at TCP's 3-way handshake	instream seq. numbers
odelta_seq	at TCP's 3-way handshake	outstream seq. numbers
stable_seq	from St. Buffer's callback	from instream packets
server_seq	outstream seq. numbers	instream seq. numbers
syscall_id	at sys. call	total # of sys. calls
unstable_syscalls	at sys. call & St. Buffer's callback	# of unstable sys. calls

These key variables include when they are set and what quantities in the system they are related to.

that the server has received from the client. During recovery, this value can be computed from the data stored in the stable buffer.

—*server_seq*: the highest sequence number of the outstream acknowledged by the client and seen by the SSW. This value also can be computed during recovery from the data stored in the stable buffer.

FT-TCP maintains the following variables for each thread of execution of the server. (A single-threaded server has a single instance of each of these variables.)

—*syscall_id*: the count of the number of system calls made by the thread. This value servers as an identifier for system calls. With this ID, calls made by the primary and the backup can be matched.

—*unstable_syscalls*: the count of the number of system calls whose records have not been acknowledged by the stable buffer. If *unstable_syscalls* is 0, then the stable buffer has recorded the results of all prior system calls.

The six variables introduced so far are summarized in Table I, which states when the variables are initialized and possibly updated by FT-TCP (through a process that will be explained in the following sections) and whether they refer to instreams or outstreams or system call counts.

To make the description of FT-TCP inner workings more precise (and, in particular, to demonstrate that hot and cold replication schemes are based on the same mechanisms) we group the actions of the system components (for each replicated service) into three modes.

(1) STANDBY MODE. In this mode the wraps are idle, while the stable buffer performs logging. Cold backups are started in STANDBY MODE and stay in it until either they are promoted or they are reconfigured to be a hot backup. If there is no need to recover a connection during its lifetime, a cold backup leaves STANDBY MODE when the server process terminates.

(2) RECORD MODE. In this mode the SSW sends incoming packets to the stable buffer. The NSW does the same with the results of system calls invoked by each thread of the application (every thread has its own queue). Every attempt by a thread to send data to the client is stalled until all its system calls are stable (i.e., until *unstable_syscalls* is 0). If the backup has failed, these output commit stalls do not occur.

Fortunately, it is not also necessary to block outgoing packets waiting for the stable buffer to acknowledge the data they are acknowledging. FT-TCP leverages the semantics of TCP, by which data must be retained at the sender until acknowledged. By changing the acknowledgment sequence number field to acknowledge to the client only data that are stable, we have the client store segments until they are stable and yet allow outgoing data to flow unimpeded. (If the part of the stable buffer that logs SSW data, i.e., the incoming network traffic, is implemented by a component interposed on the link between the client and the replicas, then that component can omit sending acknowledgments by delaying outgoing network traffic until the related incoming traffic is stable.)

The primary is in RECORD MODE until either it fails or the server process terminates normally.

(3) PLAYBACK MODE. Upon entering this mode, the backup spawns its own copy of the server process and provides this process with data that it retrieves from the stable buffer. When a thread in the backup makes a system call, a corresponding record of the primary's system call is removed from the stable buffer for ensuring deterministic execution. When the primary process accepts a connection, the backup's SSW spoofs connection establishment on behalf of the client by simulating an internal three-way handshake. When the backup process requests data from the network, the data are removed from the corresponding segment in the stable buffer and returned with the call. When the backup process wishes to send data, the segments are buffered and, if the client acknowledges that data, quietly discarded.

In the hot-backup scheme, all backup replicas start in PLAYBACK MODE; in the cold-backup scheme, a backup replica enters PLAYBACK MODE after detecting a failure of the primary. This mode ends either when the connection terminates normally or when a backup replica is promoted (i.e., switched into RECORD MODE).

As noted before, only the primary replica may be in RECORD MODE at any given time. This is to ensure that all communication with a client occurs through a single connection endpoint. Any number of backup replicas may be in either one of the other two modes.

The state transition diagrams for the modes of FT-TCP replicas (cold and hot) are shown in Figure 2.

4.2 Record Mode

In RECORD MODE the behavior of the primary process is recorded in the stable buffer so that a backup process can be made to behave the same way. The primary replica enters RECORD MODE as soon as the server process is initialized (by intercepting exec calls systemwide). Before a single instruction in the process is allowed to execute, the NSW sets *syscall_id* and *unstable_syscalls* to 0 for the original thread.

During the TCP three-way handshake, the SSW records to the stable buffer both the client's and the server's initial sequence numbers. The SSW delays the server's TCP segment that acknowledges the client's SYN packet until

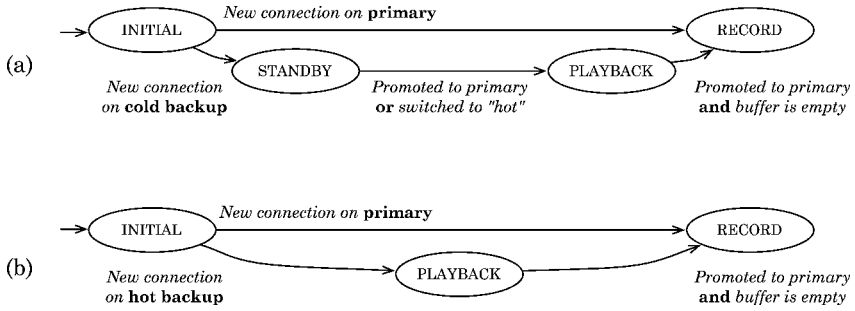

Fig. 2. State transition diagrams for FT-TCP replica modes under the cold-backup scenario (a) and the hot-backup scenario (b). The connection can be closed in any of the modes.

acknowledgment of these sequence numbers from the stable buffer. Without this precaution, an early failure might admit the possibility of a client being aware of an established connection, while a recovering replica might not know the connection exists. Finally, the SSW completes the initialization of FT-TCP by setting *idelta_seq* and *odelta_seq* to 0 and *stable_seq* to the client's initial sequence number plus one.

The primary replica leaves RECORD MODE either when the client connection is terminated properly or when the replica itself fails. In the latter case, one of the backups is elected to handle the connection. As that backup completes the failover procedure, it switches from the PLAYBACK MODE into the RECORD MODE. Variables *syscall_id*, *unstable_syscalls*, and *stable_seq* are unaffected by this transition, whereas the two sequence number deltas are updated as described in Section 4.3.

SSW in record mode. In RECORD MODE, the SSW responds to three different events: receiving a packet from the network on its way to the TCP stack, receiving a segment from the TCP stack on its way to the network, and receiving an acknowledgment from the stable buffer. The first two events are illustrated in Figure 3, where each arrow represents a packet containing a TCP segment and *seq*, *ack*, and *win* indicate the values of the sequence number, acknowledgment number, and window size for the segment.

When the SSW receives a packet from the network, it immediately forwards a copy of the packet to the stable buffer. The SSW then subtracts *odelta_seq* from the ACK number and subtracts *idelta_seq* from the sequence number. These operations change the payload, so the SSW recomputes the TCP checksum on the segment. Recomputing the checksum is not expensive: It can be done quickly given the checksum of the unchanged segment, the old sequence number, and the new sequence number [Rijsinghani 1994]. The SSW then passes the result to the server TCP/IP stack. This may be done without waiting for an acknowledgment from the stable buffer indicating that the packet has been logged.

When the SSW receives an acknowledgment from the stable buffer for a packet, it updates *stable_seq* if necessary. Specifically, if the stable buffer acknowledgment is for a packet that carries client data with sequence numbers

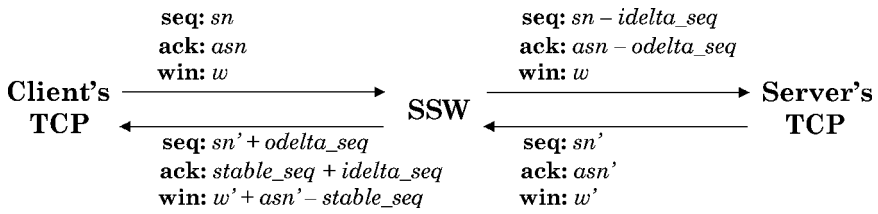

Fig. 3. Record mode operation of the SSW.

from sn through $sn + \ell$, then *stable_seq* is set to the larger of the current value of *stable_seq* and $sn + \ell + 1$.

When the SSW receives an outgoing segment from the TCP layer, it remaps the sequence number by adding *odelta_seq* to it. The SSW then overwrites the ACK number with *stable_seq*. Since *stable_seq* never exceeds an ACK number generated by the TCP layer, modifying the ACK number may result in an effective reduction of the window size advertised by the server. For example, suppose that the segment from the TCP layer has an ACK number *asn* and an advertised window of w. This means that the server's TCP layer has enough buffer space available to hold client data up through sequence number $asn + w - 1$. By setting the ACK number to *stable_seq* the SSW effectively reduces the buffering for client data by $asn - stable\_seq$. To compensate, the SSW increases the advertised window by $asn - stable\_seq$. Again, after modifying the TCP segment, the TCP checksum must be recomputed. Finally, the TCP segment is passed to the network.

NSW in record mode. When in RECORD MODE, the NSW is activated on every system call and also when a system call acknowledgment from the stable buffer arrives.

The NSW oversees the execution of each system call. Upon completion, the NSW sends the system call record (which includes *syscall_id*, system call parameters, and its result) to the stable buffer and increments both *unstable_syscalls* and *syscall_id*. The message content of a network read, however, is not sent because the stable buffer already has this data in the form of client's packets that are logged by the SSW. Furthermore, message content of a network write is not sent because backup processes will generate an identical message on their own. A short hash of the message can be sent to the backup for comparison, as a safety check against divergence of execution paths.

When the NSW receives an acknowledgment from the stable buffer, it decrements *unstable_syscalls*. And, for each write or send,[1] the NSW blocks in an output commit stall until *unstable_syscalls* is 0.

4.3 Playback Mode

In PLAYBACK MODE a backup process is driven by the NSW (using the records from the stable buffer) down the same execution path that the primary took. A replica enters PLAYBACK MODE either at connection establishment time (for a hot

[1]These are system calls that affect the client's environment.

backup) or as part of the failover procedure (of a cold backup).[2] Initially, FT-TCP sets *stable_seq* and *server_seq* to the values obtained from the stable buffer, and sets *unstable_syscalls* to 0.

Then, FT-TCP simulates a connection establishment for the TCP stack. This is accomplished through the SSW, which can both create and respond to the segments required for the TCP three-way handshake. First, the SSW creates a SYN packet that appears to be from the client (i.e., it has the client's IP as a source address) and has an initial sequence number of isn_c, as was chosen by the client in the three-way handshake with the primary. SSW captures the response from the TCP and saves the initial sequence number chosen by the server's TCP in isn_s. Discarding the outgoing segment, the SSW finally creates an acknowledgment that appears to be from the client and passes it to the TCP stack.

When the three-way handshake simulation is complete, the TCP stack has what it believes is an open connection in which client data is expected to begin at sequence number $stable\_seq + 1$.

SSW in playback mode. FT-TCP feeds network data to the backup replica directly through the NSW (unlike the system of Koch et al. [2003], which injects the packets into the TCP stack). So, on the backup, both the SSW and the TCP stack are idle during normal operation; they consider the connection open, but there are no bytes going through. This design decision has two advantages. First, it is efficient, since the data takes the shortest path to its destination: from the stable buffer directly into the application's buffer. Second, it prevents the TCP stack from estimating the packet round-trip time incorrectly, which can lead to poor performance upon failover. Of course, since the TCP stack is not involved, the task of reassembling packets is left to the stable buffer. Fortunately, that's the only task of TCP that we must handle; flow control, congestion control, and retransmission are done by the TCP on the primary.

NSW in playback mode. When a backup process in PLAYBACK MODE makes a system call, the NSW uses the corresponding system call record from the primary to do one of several things:

—For calls that query the operating system environment, such as getuid, getpid, and gettimeofday, the backup immediately returns the result that the primary got.
—For a send, the backup queues the data passed by the server application in a (local, in-memory) *send buffer* and returns the result that the primary got. The send buffer is needed on the backup so it can resend to the client any outstream data that got lost in the crash. Data are removed from the send buffer as client acknowledgments arrive on the backup. For debugging, the

[2]It is also possible to envision a hybrid solution, wherein backups start in STANDBY MODE and then, based on elapsed time, network conditions, or some other observable metric, become hot and switch into PLAYBACK MODE. This might allow the system to avoid the increased overhead of hot backups in the case of short-lived connections, but still achieve faster failover for long-lived connections. We have not explored this idea in practice.

messages returned by the backup and the primary (or their checksums) can be compared to flag any inconsistencies.

—For a recv, the backup waits until all necessary data packets are in the stable buffer, copies the same number of bytes as the primary got, and returns the same result.

—For the two calls that return socket status, namely select and poll, the backup returns the value from the primary (if a timeout was specified, then the backup invocation will block until the same call on the primary times out).

—For all others, the backup executes the call and compares its result to what the primary got. Any inconsistencies are flagged as a potential divergence in execution paths.

The first category of calls resolves simple sources of nondeterminism such as different clock values on the replicas and different attributes of their process environments. Special treatment of send and recv allows us to pass client data efficiently from the stable buffer directly into the application, without having to feed the packets through the backup's TCP. (This means that as far as the backup's TCP is concerned, the connection to the client during this period is idle.)

If an invocation on the primary returned an error code, it is important to return the same error code on the backup, as we do for select and poll. In particular, if a nonblocking read returns an error indicating the lack of any data to return, it is important to return the same error on the backup even if packets with new data have arrived by the time this system call is invoked on the backup.

All system calls not mentioned explicitly can be complex in their semantics and side-effects. We consider these general cases of nondeterminism outside the scope of this article. We allow these system calls to execute on the backup checking for the same results as the calls on the primary. These system calls returned identical results for the three applications that we considered.

4.4 Termination and Failover

When the primary server process shuts down cleanly, the backup replicas in PLAYBACK MODE eventually shut down, too; but if the backup detects a failure, then it switches from PLAYBACK MODE to RECORD MODE.

Failures are detected as follows: Every time a message is received from the primary, the stable buffer resets the failure detection timer. If no requests arrive for a prescribed time interval, the stable buffer sends a heartbeat probe that the primary is expected to acknowledge. The idea is to avoid the overhead of heartbeats when the buffering traffic is enough to indicate liveness. If the primary does not acknowledge the heartbeat probe within a certain time, the backup assumes that the primary has failed and initiates failover.

In such a situation, the backup first executes all the system calls that the primary had executed before failing, consuming all buffered instream data. It

Table II. Summary of the Activities of the Two Wraps in the Three Replica Modes

Host type:	Cold backup	Hot or cold backup	Primary
Mode:	STANDBY →	PLAYBACK →	RECORD
NSW	inactive	feeds logged data to application	logs syscalls, blocks for output commit
SSW	inactive	forges connection to server, keeps it alive	logs instream data, translates seq nums

The arrows show possible mode transitions for cold-backup and hot-backup schemes. When the primary host is the original and not a failed over backup, the sequence number translation is a no-op.

then sets the delta values as follows.

$$idelta\_seq = stable\_seq - isn_c - 1 \quad (1)$$

$$odelta\_seq = server\_seq - isn_s - 1 \quad (2)$$

Sequence numbers in *stable_seq* and *server_seq* identify the first unacknowledged bytes in the instream and outstream, as seen by the client's TCP stack. Incrementing by one each of *isn*s we get the sequence numbers of the first byte in the instream ($isn_c + 1$) and outstream ($isn_s + 1$) as seen by the server's TCP stack. (Adding one compensates for the one byte taken up by the SYN segment.) The deltas for each direction are set to the difference between the client's and the server's sequence numbers (e.g., $idelta\_seq = stable\_seq - (isn_c + 1)$). Subtracting the appropriate deltas from the sequence numbers in the instream packets and adding them to the sequence numbers in the outstream packets (as shown in Figure 3) has the effect of converting the sequence numbers between the view of the client and the view of the newly promoted primary. This completes the failover and the replica moves into RECORD MODE.

The activities of the wraps in the three modes are summarized in Table II.

5. NONDETERMINISM

For replicas to execute deterministically during playback, it is not enough to ensure identical input. Anything that changes the state of a server, such as error conditions and asynchronous events, needs to be delivered consistently, too. The most immediate sources of nondeterminism arise from the system calls that depend on the status of a socket, namely select and poll.

For example, suppose a poll on the primary indicates that there is data to be read; the primary then would proceed and read the data. But if at the same point poll at the backup shows no data, the backup may yield the CPU to a different thread and follow a different execution path. Therefore, we forward the results from select and poll from the primary to the backup to ensure deterministic reexecution.

Like poll, a nonblocking read can indicate the lack of data in a socket buffer (by returning -1 with *errno* set to EAGAIN). We use the term *readlength* to refer to a result returned by a read. We address this source of nondeterminism by forwarding readlengths from the primary to the backup to ensure deterministic

reexecution. Although we found the readlength of -1 (i.e., the error condition) to be a source of nondeterminism, returning the same data in chunks of different sizes (e.g., chunks of size 4 and 5 bytes on the primary and 2, 3, and 4 bytes on the backup) did not result in divergent execution for the applications that we tried. We conjecture that most applications do not depend on the particular number of bytes returned by a read because they invoke data processing code not upon every read, but upon reaching record boundaries of their application-level protocol. However, if one writes an application that is sensitive to the particular number of bytes, FT-TCP can be configured to preserve the readlengths across replicas.

Other system calls are important sources of nondeterminism. For example, when several processes compete for a file lock, there is no guarantee that they will acquire it in the same order on the primary and on the backup. Hence there may be processes for which lock acquisition will succeed on the primary, but will fail on the backup or vice versa. For some applications (the ones written to retry lock acquisitions indefinitely) this may not pose any problems. But for others, all lock requests must return the same results on both replicas. Therefore, we intercept the system call implementing file locking operations (fcntl on Linux) and enforce identical order of acquisitions on the replicas.

Thread scheduling and signal handling are both commonly identified as sources of nondeterminism. Neither proved to be problematic for the three services that we evaluated. Of course, services like Samba use signals (we verified this by looking at the source code), but either signals were rare, or nondeterministic reordering of signals did not cause backup servers to diverge from the primary. Building a commercial fault-tolerant TCP system would require capturing and replaying signals at the appropriate times in the execution path [Bressoud and Schneider 1996; Slye and Elnozahy 1996] and implementing efficient deterministic thread scheduling [Basile et al. 2003; Napper et al. 2003].

Finally, we had to address the nondeterminism introduced when a server generates a random value and then uses it in communications with the client. Section 7 shows how we modified the server applications to ensure that identical random values are generated on the primary and on the backup. To avoid source code modifications, we have explored using a protocol-specific "hook" to capture randomly generated values and make the appropriate substitutions [Zagorodnov and Marzullo 2005].

Conceptually, there is a spectrum of servers in terms of how sensitive their network output is to nondeterminism inherent in their implementation or in the system. At one end are fully deterministic servers, such as a Web server relaying static content; at the other end are servers that exhibit nondeterminism throughout the lifetime of a connection. In the middle are servers that behave nondeterministically during connection initialization (e.g., due to generation of nonces and IDS), but not in the steady state. FT-TCP can replicate the latter and the fully deterministic servers (and does so efficiently, as the experients will show).

6. STABLE BUFFER

In this section, we describe the stable buffer protocol and how it can be optimized for performance.

6.1 Logging Protocol

In our implementation both wrappers on the primary communicate with the stable buffer using a TCP connection. On the backup, the wrappers use (kernel-level) function calls. The stable buffer protocol is based on pairs of request-reply messages that contain a header that is optionally followed by data. The header includes message type, several identifiers for quickly finding the appropriate queue for that message, and metadata such as sequence numbers and system call index. In our implementation the request header is 62 bytes long and the reply header is 39 bytes long.

Requests *write_packet*, *write_isn*, and *write_syscall* are issued by the wrappers in RECORD MODE to place information in the stable buffer. The buffer replies with simple acknowledgments, in the form of a stable sequence number or a latest *syscall_id*. Requests *read_data* and *read_syscall* are issued in PLAYBACK MODE and cause the stable buffer to reply with the corresponding information and optionally remove those records. With only one backup we remove the records immediately, but with multiple backups buffer content must persist in the stable buffer until all backups have had a chance to read it. Since client data are stored in the stable buffer as packets, to service a *read_data* request the stable buffer may have to remove contents of multiple packets and fuse them together into a message of the same size as was returned on the primary.

Each *write_packet* reply from the stable buffer is essentially a sequence number: It is the lowest sequence number of client bytes that are not logged. Thus, the sequence of acknowledgments is monotonically increasing (ignoring the 32-bit wrap). This means that the last acknowledgment in any batch contained in a segment is the only one that needs to be processed by the SSW, since it dominates the other acknowledgments. We have found, though, that the overhead incurred by having the SSW process each acknowledgment is small enough that it is not worth taking advantage of this observation.

6.2 Ack Strategies

As described in Section 4.2, the SSW ensures that a client does not discard instream data before the SSW knows they are logged in the stable buffer. This is done by setting the ack field of outstream segments to *stable_seq*. If all outstream segments are thus modified and sent to the client but no additional segments are generated by the SSW, the connection may be unable to reach the maximum possible throughput. To make this concrete, imagine a segment arriving at the SSW from the client carrying bytes ending with sequence number *sn*. The SSW sends this data to the stable buffer, but by the time the server's TCP layer generates an ack for them, the stable buffer has not yet acknowledged them, so *stable_seq* is still less than $sn + 1$. Even if the acknowledgment from the stable buffer arrives immediately thereafter, the client's TCP layer will not become aware of it until the server's TCP layer sends a subsequent segment.

Such a situation inhibits the client's ability to measure the round-trip time (RTT), which is used by TCP to set the retransmission timeout (RTO), which is crucial for congestion control and for reliability. A worse situation occurs, however, when the outstream traffic is low and the instream traffic is blocked

because of windowing restrictions. For example, consider what happens when slow start [Jacobson 1988] is in effect. Suppose that the client sends two segments S_1 and S_2 when the client's congestion window is two segments in size and is less than the server's advertised window. If the acks to these packets are generated before either are logged in the stable buffer, then the client will block with a filled congestion window, and the server will block, starved for data. This situation will persist until the client's TCP layer retransmits S_1 and S_2.

We experimented with three simple ack strategies that avoid such problems. All three suppress outstream segments that carry no data and do not ack additional data, since such segments do not affect the connection dynamics. The three strategies differ in when they generate new outstream acks in response to acknowledgments from the stable buffer.

—*Lazy*. The SSW generates an ack for a segment S if S was the most recent segment that the server's TCP has acked.
—*Delayed*. The SSW generates an ack for a segment S if the server's TCP layer has acked S at any point in the past.
—*Eager*. The SSW generates an ack for every acknowledgment it receives from the stable buffer (unless that packet has already been acknowledged to the client), thus potentially acking every instream packet.

Lazy generates the smallest number of acks necessary to keep the connection active without retransmissions due to stalling. This count of acks can be smaller than the number of acks sent from the TCP stack down to the SSW; multiple outgoing acks may get merged into one actual ack since the SSW only considers the latest one. *Delayed* is equivalent to delaying the outgoing ack segments until *stable_seq* catches up to their ack sequence number, so it can be thought of as regular TCP behavior with some additional latency on every outgoing ack packet. Finally, *Eager* acks every packet, generating considerably more packets than the TCP layer. In contrast, typical TCP implementations ack at most every other packet. The motivation behind *Eager* is that these additional acks can keep the client's TCP more up-to-date about socket buffer space available on the server. All three strategies are illustrated in Figure 4. We show in Section 8 how these strategies perform in practice.

6.3 Nagle's Algorithm

Since the exact timing of acknowledgment packets from the stable buffer may affect the dynamics of the client connection, a relevant question is whether Nagle's algorithm [Nagle 1984] should be disabled for the interreplica TCP connection (i.e., between primary and backup). When Nagle's algorithm is disabled, every message is placed in a segment and sent as soon as possible, allowing for the fastest possible update of *stable_seq*. With Nagle enabled (which is the default TCP setting) small messages from the application are delayed slightly in hope of batching them together with other small messages and reducing the total number of segments on the wire. While this conserves bandwidth, it also increases latency. We explore this trade-off in Section 8.

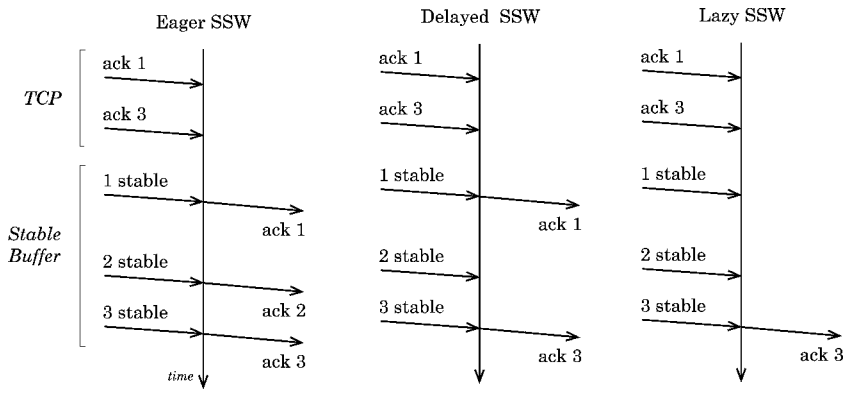

Fig. 4. Behavior of the three SSW acknowledgement strategies, given the same sequence of inputs: two acknowledgement packets from TCP, which are suppressed by the SSW, and three updates from the stable buffer.

7. APPLICATIONS

We selected three popular TCP-based servers to study the difficulties of replicating real applications and to measure the performance overhead imposed by FT-TCP: the *Darwin Streaming Server* (DSS) that serves multimedia content, the *Samba* server that implements Microsoft's file and printer sharing protocols, and the *Apache* Web server. Besides their popularity, these applications were attractive to us because they tend to have long-lived connections (which are worth recovering) and their source code is publicly available. Furthermore, each one handles connections differently: Samba spawns a separate process for each client connection, Apache assigns an incoming connection to an already existing idle process, and DSS handles all connections asynchronously, in a single thread, so three common types of network programming practices are represented by these applications. We discuss such structural details next in the sections dedicated to the individual servers.

Another application that we used for studying the effect of FT-TCP on throughput is *ttcp*, a simple bandwidth testing tool available freely since early 1980's. All data in ttcp are fabricated by the sender and thrown away by the receiver, allowing the connection to fully saturate the link. In our experimental setup (described in Section 8), ttcp can obtain over 99% of maximum theoretical throughput of TCP/IP on Ethernet configurations up to a gigabit per second.

7.1 Darwin Streaming Server

DSS is available under an open source software license from Apple Computer, Inc. Although it is generally considered better to stream multimedia over datagram-based protocols like UDP, streaming is frequently done over TCP to bypass firewalls. In both cases the stream is encapsulated inside the Real-Time Streaming Protocol (RTSP).

DSS runs as one process with at least three threads: one for all network communication, one for servicing requests, and one auxiliary thread. The application is event-driven and all I/O is done asynchronously. For each viewing session

there are two connections: one for controlling the stream and one for the stream itself. The streams live at least as long as they are being played, and the connection state indicates the position in the stream. Hence, if a failure causes the connection to fail, then the client needs to reopen the connection and reposition the playback point in the stream. Our viewer has application-level recovery: It remembers where the playback of the stream left off and repositions for the client when the "play" button is pressed again.

DSS is an interesting service to consider because it uses multiple connections per client and also because it is a multithreaded application. It has some attributes that make it less challenging. In particular, it only reads files, making the output commit problem an issue only with the playback of the stream. Additionally, it generates a large amount of output data in response to small requests, thus reducing the load on the buffering mechanism.

We ran an unmodified version of DSS on top of FT-TCP to explore its sources of nondeterminism. The NSW detected a nondeterministic divergence between the primary and backup almost immediately. This nondeterminism occurred when the server generated a random *Session ID* that was sent to the client in response to a SETUP request of the RTSP protocol. The ID is used for all subsequent communication in a session. If the primary and the backup generate different IDs, then all client requests will be rejected because of an invalid ID. To generate the same IDs while keeping the protocol cryptographically secure, we retained the calls to a pseudorandom number generator, but made sure that the values used to compute the seed are derived from the system calls whose return values we insert on the backup, such as gettimeofday. After we changed the source code of DSS to make sure identical IDs were generated, we saw no further execution deviations between the primary and backup servers.

7.2 Samba Server

The Samba server implements Microsoft's family of protocols for sharing files and printers, such as the Server Message Block (SMB) and the newer Common Internet File System (CIFS) [Hertel 2003; X/Open 1992]. These protocols were originally designed to run over LAN transport protocols, but these days they use TCP/IP almost exclusively.

On the Linux platform, a new Samba process is spawned by the *inetd* dæmon for each incoming connection. Connections typically last a long time: for as long as a remote file system is mounted on the client. Clients may mask connection failures if they occur during idle periods (no outstanding requests) by reconnecting to the service upon the next user command. If, however, a connection is broken during an active transfer, the transaction is abandoned and an error is raised.

We found two sources of nondeterminism in Samba. The first one has to do with the challenge-response authentication scheme used for access control, in which the server generates a random challenge string that the client encrypts with a password and passes back to the server for comparison. If the random challenges generated by the replicas differ, then the response from the client will only succeed in authentication on the primary, while the backup will reject

that connection. The second source of nondeterminism, similar in principle to the Session ID in DSS, was the generation of a file handle for each file opened by a client, who then uses it in all file operations. As with DSS, we changed the code to ensure that the same challenges and the same file handles were generated on the primary and on the backup, taking care to preserve the cryptographic integrity of the protocol. After that we saw no further execution deviations in any of our experiments.

7.3 Apache Web Server

Apache has been the most popular Web server on the Internet for many years now. It communicates with clients using a relatively simple HTTP 1.1 protocol, but it is a nontrivial application to replicate as it relies on many modules to extend its functionality.

The server uses one master thread for receiving all network requests, which are handed off to one of the idle service processes that Apache maintains. If the number of service processes is insufficient to handle the connection load, the master spawns additional ones. Apache is similar to Samba in that separate connections may be handled by different entities (processes in Samba and threads in Apache), but it is different in that Apache's threads are not spawned by an external dæmon and they typically handle many connections throughout their lifetime.

We were primarily interested in Apache for helping us understand how FT-TCP performs when many connections are created and torn down simultaneously.

8. OVERHEAD

We studied the overhead of FT-TCP with a prototype written as a kernel module for version 2.4.20 of Linux. No kernel recompilation is needed to use it; the module loads on-the-fly into a standard kernel. The SSW relies on *netfilter* hooks for intercepting packets and the NSW uses several symbols from the kernel (*sys_call_table* among them) for intercepting system calls and TCP-related functions. The FT-TCP module on the primary communicates with the backup module through a kernel-level socket, so no additional context switches are introduced.

For all experiments we used 1.4-GHz Pentium III workstations with a 512KB L2 cache and 1GB of RAM. Each machine had two on-board *Intel Pro 1000 XT* 1Gbps Ethernet adapters that we could configure to run at speeds of 10, 100, and 1000Mbps. By varying speeds and wiring configurations we experimented with five network setups: 10-10, 10-100, 100-100, 100-1000, and 1000-1000, where the first number specifies the bandwidth of the client-primary link and the second number specifies the bandwidth of the primary-backup (or interreplica) link. All machines were physically interconnected through a 100Mbps switch via one of their adapters (except in the 1000-1000 setup, where the client and the primary were connected directly), and the two replicas had a direct connection through their second adapters. All links were full-duplex.

We used packet traces from the *tcpdump* utility collected on the client machine for calculating throughput and latency. To determine the *aggregate throughput* of an incoming (client-to-server) data flow, we recorded the time

Table III. Bulk Incoming TCP Performance under 10-10 Setup and 10-100 Setup

	(*a*) 10-10 setup				(*b*) 10-100 setup			
	Throughput (*KB/s*)	% of Clean	Average Latency (*ms*)	Ack Count	Throughput (*KB/s*)	% of Clean	Average Latency (*ms*)	Ack Count
Clean	1158	100	7.37	1438	1158	100	7.37	1438
Lazy	171	15	49.88	482	1158	100	7.37	1438
Delayed	994	86	8.60	1439	1158	100	7.37	1439
Eager	996	86	8.58	2874	1158	100	7.37	1439

These setups are with tcpdump on the client by running ttcp on the two ends, with Nagle enabled on the interreplica connection.

when acknowledgment packets were received by the client and the total number of bytes acknowledged at that point. For an outgoing data flow, the sending time of server-bound acknowledgments was recorded. In both cases, the slope of a least-squares fit for this data provides an accurate representation of the steady-state throughput of a connection.

We also measured the *average packet latency* as the mean time from when a data-carrying packet departed from the client and when an acknowledgment for that packet arrived back at the client. Acknowledgments frequently ack several packets at once, so this measure should not be taken as the minimum possible round-trip time of a TCP packet.

In the "bulk throughput" tables that follow we present both throughput and latency as mean values from 15 identical experiments. Each experiment consisted of a 4MB transfer, except for the 1000-1000 setup where 40MB were sent. We used default TCP buffer sizes (8KB) for all setups except 1000-1000, where the buffer was increased to 64KB (more on this in Section 8.2). We first gathered the results of a nonwrapped TCP stack at the primary machine with the same client and server applications as used for the experimental runs. Those results are labeled throughout as *Clean*, and we regularly use the percentage of Clean throughput obtained by FT-TCP connections as the key measure of overhead.

8.1 Throughput of ttcp

Table III(a) shows how incoming ttcp transfer behaves on the 10-10 network setup under the three ack strategies. The Delayed and Eager strategies show similar performance, both obtaining 86% of Clean's throughput. We will explain in Section 8.2 why it is difficult to achieve much better throughput than this under such a *symmetric setup* where the bandwidths of the client-primary and primary-backup links are identical. Lazy only achieves 15% of Clean's throughput and the tcpdump traces show why.

With Clean, the server application is at least as fast as the client. Good bandwidth utilization is achieved through a well-formed interleaving of the instream data packets within the advertised window with the sequence of acks returning to the client. For example, suppose that the advertised window has a capacity of six packets. At some point in the steady state of the transfer, the client sends segments x, $x+1$, and $x+2$. At this point in the interleaving,

Table IV. Bulk Incoming TCP Performance of FT-TCP under Various Network Setups

Client-primary setup		Inter-replica setup					
		10		100		1000	
Bandwidth	Option	Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~
10	Delayed	**86**	77	**100**	**100**		
	Eager	**86**	77	**100**	**100**		
100	Delayed			80	77	**100**	**100**
	Eager			**85**	77	**100**	**100**
1000	Delayed					36	57
	Eager					56	**60**

These setups are varied (10-10, 10-100, etc.), as are configuration options: ack strategies (Delayed and Eager) and use of Nagle. Numbers shown are percentages of Clean throughput, that is, without FT-TCP. Bold font highlights maximum values under each network setup.

the server sends an ack for the bytes in $x-1$, which allows the client to send packets $x+3$, $x+4$, and $x+5$, and then receives the ack for the bytes in $x+2$. This pattern then repeats. Under this interleaving, the client is rarely stalled awaiting an ack from the server to allow more data to be sent.

Under FT-TCP, the Lazy ack strategy exhibits a pattern in which the client sends all the data possible in the window and then stalls for an acknowledgment. This ack is sent only after the stable buffer notifies the primary that the client's data have become stable. This pattern of behavior is indicative of a fast sender and a slow receiver (see [Stevens 1994, p. 279]). Both Delayed and Eager avoid this performance-draining pattern by acking more promptly.

When the speed of the interreplica link is increased to 100Mbps, FT-TCP seems to no longer impose any significant overhead on the connection, as shown in Table III(b). All three ack strategies are able to keep up with Clean TCP throughput (Eager even seems to exceed it, but this is not statistically significant), with similar variance, identical average packet latencies, and essentially the same number of acks. The reason is that, with a faster interreplica link, incoming packets manage to become stable before the primary attempts to acknowledge them.

Incoming ttcp throughput results for all five network setups are summarized in Table IV in terms of percentages of Clean throughput. The rows determine the speed along with the ack strategy on the client-server link, while the columns determine the speed and use of Nagle algorithm on the inter-replica link. For example, the 10-100 results may be found in the first and second rows, labeled "10," and the third and fourth columns, labeled "100." For each setup there are four measurements, allowing us to identify the best parameters for that setup. The highest numbers within each setup are set in bold. Some cells are empty because the setup is not sensible (e.g., client link faster than the interreplica link).

It is evident from the table that *asymmetric setups* (10-100 and 100-1000) impose practically no overhead on ttcp, whereas symmetric setups suffer a penalty of 15–40%. However, the 40% loss took place under a challenging setup in which

a single client was able to saturate a 1Gbps link with incoming data. We expect many of the real-world network configurations to be asymmetric.

These results also indicate that it is best to keep Nagle's algorithm on, since turning it off either lowers throughput or doesn't make any difference. One exception to is in the 1000-1000 numbers, to be discussed shortly. However, we defer the general discussion of Nagle's algorithm until Section 8.3, which describes more realistic applications.

We additionally performed outgoing transfers with ttcp and found that FT-TCP did not add any significant overhead under any network setup. This is not surprising, since outgoing data is not sent to the stable buffer and can be sent out immediately. As described in Section 4.2, write may block waiting for system calls to become stable, but since here data is written in big chunks, this overhead is negligible.

8.2 1Gbps Experiments

The goal of this section is to explain the performance under the 1000-1000 network setup and to give intuition for why all symmetric setups are bound to suffer some throughput loss.

To saturate a 1Gbps link we increased the Ethernet frame size from the standard 1500 bytes to 9000 (the so-called *jumbo frame*). We also configured ttcp to use the largest possible TCP buffer size of 64KB on the receiver. (Larger buffers can be used together with the *window scale* option of TCP, but our implementation does not support that option.) With this buffer size and the maximum segment size of 8960 bytes, which is 9000 minus 40 to account for the TCP and IP headers, our TCP stack advertises a window of 53720 bytes, which is large enough for exactly 6 packets.

For Clean runs, the client never sends more than 4 packets before it gets an ack from the server, so the pipe is always full of data and the sender never blocks waiting for the receiver. With FT-TCP under 1000-1000, as well as under other symmetric setups, every packet travels twice as far (first from the client to the primary, then at the same speed from the primary to the backup) so we can expect the round-trip latency to double at least. As the latency doubles, so does the bandwidth-delay product (i.e., the size of the pipe) and it now takes twice as many packets to keep the connection going. So instead of the 4 outstanding packets we saw under Clean, we may expect up to 8 with FT-TCP. The problem is that the maximum number of packets that our window allows is 6, so occasionally the client has to stop sending and wait for an ack.

This is, indeed, what we see in the tcpdump traces. Luckily, there is some overlap between actions of TCP and FT-TCP. The minimum acknowledgment latency for a TCP packet on a 1Gbps link is about 372 μsec of which, in theory, only 72 μsec are spent by the packet on the wire. When we measured the average time it took for our stable buffer to acknowledge a packet, we obtained a very similar value of 378 μsec. Because the server's TCP is processing the packet in parallel with the copy of the packet traveling to the stable buffer, the overall packet latency seen by the client does not quite double with FT-TCP: It is 686 μsec.

Table V. Bulk Incoming Samba File Transfer Performance of FI-TCP under Various Network Setups

Client-primary setup		Inter-replica setup					
		10		100		1000	
Bandwidth	Option	Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~
10	Delayed	55	**75**	61	**100**		
	Eager	55	**75**	61	**100**		
100	Delayed			15	78	15	**95**
	Eager			15	**79**	15	**95**
1000	Delayed					1	38
	Eager					2	**63**

These setups vary (10-10, 10-100, etc.) as do configuration options: ack strategies (Delayed and Eager) and use of Nagle. Numbers shown are percentages of Clean throughput, that is, without FT-TCP. Bold font highlights maximum values under each network setup.

In particular, this means that the time it takes to make a copy of a packet (about 25 μsec) is absorbed by the overlap.

8.3 Samba Throughput

The first real application that we studied under FT-TCP was Samba. Our bandwidth experiments consisted of logging into the server and performing a single *put* or *get* operation to transfer a 4MB (or, on a 1Gbps link, a 40MB) file to or from the server. The throughput percentages of incoming transfer experiments are summarized in Table V.

Just like ttcp, Samba runs best on asymmetric setups. It reaches 100% of Clean throughput with 10-100 and 95% with 100-1000. We were initially surprised because the application is much more complex (i.e., has many more system calls to log). Throughput loss with symmetric setups ranges from 25–37%. The latter number, derived from the bottom right cell, is comparable to the 40% loss suffered by ttcp under 1000-1000 setup. In both cases, Eager with Nagle disabled seems to yield the best throughput.

The most pronounced difference between ttcp and Samba is in the effect of Nagle's algorithm on throughput. Recall that for ttcp Nagle worked best, but with Samba it consistently leads to lower throughput and, in fact, produces increasingly disastrous results as the speed of the client link increases, dropping throughput to around 15% of Clean with a 100Mbps and as little as 2% on a 1Gbps link! By examining tcpdump traces, we determined that the root of the problem is that Samba sends data in batches (of about 64KB), with an acknowledgment expected after every batch. The speed of the transfer is affected by how promptly the server can send an acknowledgment. Because Nagle's algorithm can slightly delay responses from the stable buffer, this can, in turn, enlarge the time it takes for the Samba server to send an acknowledgment, since all send calls must wait for the preceding system calls to become stable. Batch after batch, these delays add up.

The performance of outgoing Samba transfers is summarized in Table VI. Because Samba uses many send calls, all of which can block in an output commit stall, no network setup reaches 100% of Clean throughput, but many come close.

Table VI. Bulk Outgoing Samba File Transfer Performance of FI-TCP under Various Network Setups

Client-primary setup		Inter-replica setup					
		10		100		1000	
Bandwidth	Option	Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~
10	Delayed	42	**98**	42	**99**		
	Eager	42	**98**	43	**98**		
100	Delayed			7	**93**	7	**96**
	Eager			8	92	8	94
1000	Delayed					3	**71**
	Eager					3	**71**

These setups vary (10-10, 10-100, etc.), as do configuration options: ack strategies (Delayed and Eager) and use of Nagle. Numbers shown are percentages of Clean throughput, that is, without FT-TCP. Bold font highlights maximum values under each network setup.

In fact, most setups are only short by 4% or less and only 1000-1000 loses 29%. This is in contrast to ttcp, where no overhead was measured on the outgoing transfers because ttcp sends data in just several sends. As with the incoming transfers, it is better to turn off Nagle's algorithm. There isn't much difference between Delayed and Eager, which is to be expected since ack strategies only matter when there is incoming data to be acknowledged.

8.4 Logging Cost

In this section, we compare hot and cold backups in terms of throughput, as well as consider the cost of intercepting system calls.

As discussed in Section 4.2, in RECORD MODE both the NSW and the SSW buffer incoming packets and system call results. The difference between hot and cold backups is that in the former case the buffered records are consumed promptly, while in the latter they keep accumulating in the stable buffer. For some applications, buffering all system call results may be unnecessary, so it is worthwhile to consider the cost of buffering just packets and readlengths. So, in Figure 5 we show average throughput of an incoming Samba transfer on a 100-1000 setup with hot and cold replicas. Both are divided further into three modes.

—*Packets, readlengths, and system calls* are recorded in the stable buffer. This is the full-fledged mode of FT-TCP operation that allows replication of arbitrary programs. It is also the most costly one. All throughput numbers in previous sections were obtained in this mode.

—*Packets and readlengths* are recorded, but not system calls, allowing us to determine their contribution to overhead. If an application can run deterministically without interception of system calls then this mode is sufficient for correct operation.

—*Immediate*: Packets and readlengths are recorded, but FT-TCP does not perform output commit stalls. In this mode recovery cannot be guaranteed and it is only useful for the purposes of evaluating the minimal overhead imposed by FT-TCP's interception and buffering mechanisms.

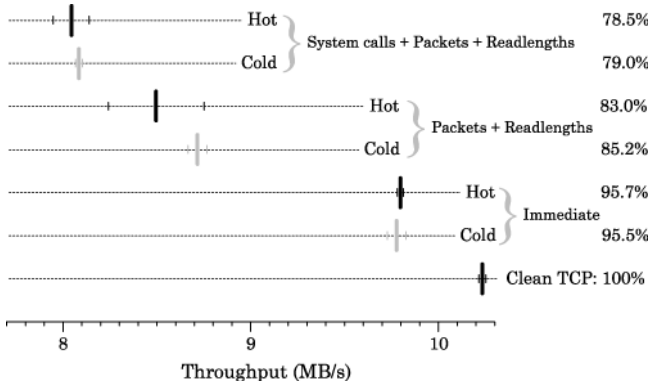

Fig. 5. Bulk throughput of an incoming Samba transfer under 100-1000 setup with different levels of logging: interception of network packets and readlenghts without (Immediate) and with output commit, as well as maximum interception with output commit, including system calls.

The first observation to make from this bar chart is that the values in each pair of hot and cold measurements are very close. As expected, cold sometimes performs slightly better, but not by much. This implies that the active backup process is not significantly affecting the operation of the stable buffer. Neither is the buffering affecting the speed of the process, because we did not see any increases in the average size of the system call queue (if the backup process were lagging behind the primary, then its queue of system calls would keep growing).

As for overhead, roughly a quarter of it (5% of Clean performance in this case) is introduced by our interception mechanism, as shown by difference between the Immediate and Clean values. Then, about half of the overhead (10% of Clean) stems from buffering of packets and readlengths. And, finally, the remaining quarter is due to system call interception and buffering. This indicates that buffering systems calls is not the major cause of overhead in applications like Samba. As expected, all these types of overhead are much less pronounced for simpler applications like ttcp.

8.5 Latency

For interactive services, such as a terminal connection, responsiveness of the server may be more important than its maximum bandwidth. To see how FT-TCP affects latency characteristics of services, we executed short requests to a Samba server under the 10-100 setup and analyzed client-side packet traces for these connections. Each instance of the experiment (a directory listing request) consisted of an 87-byte request, a 464-byte reply with the directory contents, a 39-byte server status request, and a corresponding 49-byte reply. We defined *Samba request latency* as the time interval between the client sending a 87-byte request and the client receiving the 49-byte reply. We also measured the *average* TCP *packet latency* of all incoming data-carrying packets as the time between the moment the packet left the client and the moment the packet acknowledging that data arrived at the client. Finally, for the runs done under

Table VII. Latencies Incurred while Making "Short" Requests to a Samba Server under the 10-100 Network Setup

	Samba Request			TCP Packet			Buffering		
Latency (ms)	*min.*	***avg.***	*max.*	*min.*	***avg.***	*max.*	*min.*	***avg.***	*max.*
Clean	2.11	**2.18**	2.97	0.24	**0.70**	1.92			
Cold	5.39	**5.82**	6.99	0.71	**2.05**	4.25	0.05	**0.52**	1.62
Hot	5.80	**6.18**	7.33	0.75	**2.23**	4.33	0.04	**0.55**	3.69

Samba Request refers to the overall round-trip latency for a pair of small Samba requests, *TCP Latency* is the average time it took for the server's TCP to acknowledge the packets carrying such requests, and *Buffering* measures how long on average it took for the backup to acknowledge stability of the data in such requests.

FT-TCP, we measured the internal *buffering latency*, which is the time elapsed between a buffering request and a reply as measured on the primary.

Results of these latency experiments are shown in Table VII with minimum, mean, and maximum values. There were 30 Samba request latency measurements, 68 packet latency measurements, and 230 buffering latency measurements (which include both packet and system call requests). The two highlighted pairs of mean values are not significantly different when their expected variance is calculated at the 95% confidence interval.

As the second column of the table illustrates, the average Samba request latency almost tripled (from 2.18 ms to 5.82 ms under cold and 6.18 ms under hot) when FT-TCP was added. On the one hand, the latencies under FT-TCP are still small enough that the overhead of replication may be unnoticeable to a human user issuing commands sequentially, especially when the connection traverses a WAN with latencies one or two orders of magnitude larger. On the other hand, the overhead of 180% *can* affect throughput and maximum request rate of intensive workloads, as will be shown in Section 8.6.

The increase in Samba request latency can be attributed to the increase in TCP packet latency. Average packet latency, shown in column five, also roughly tripled from 0.7 ms to around 2.2 ms because of interception and buffering overhead. The packet latency is not directly comparable with the Samba request latency because a Samba request consists of two incoming and two outgoing data packets along with some acks, but the two latencies are correlated. For transfers that saturated the link and used mostly full-sized packets (1460 data bytes), such as *ttcp in* and *Samba in*, the latency of packets for both Clean TCP and FT-TCP connections on the 10-100 configuration is around 7.4 ms, as was shown in Table III.

There is no statistically significant difference between hot and cold average packet latencies; the same is true for the FT-TCP buffering latencies. Because the distribution of values in all of these experiments is not perfectly normal (three different types of packets produced a trimodal distribution), the averages and their confidence levels are not ideal for understanding these data. Still, since both minimal and maximal values of hot are higher than the corresponding values for cold, we can conclude that the increased buffering latency is the cause of the higher hot Samba request latency when compared to cold.

Table VIII. Average Bulk Client Throughput and Average Packet Latency for Multiple Simultaneous ttcp Connections (2–128)

		2	4	8	16	32	64	128
Throughput	*Clean*	5794	2850	1456	728	363	182	94
(KB/s)	*FT-TCP*	5793	2843	1452	728	363	182	93
Latency	*Clean*	0.76	0.88	1.010	1.161	2.790	8.059	19.211
(ms)	*FT-TCP*	0.80	0.95	1.030	2.253	4.701	11.355	21.043

The minimal and maximal buffering latencies are also useful measures of the range of the round-trip times for messages between our replicas. The RTT is useful for determining reasonable values for the failure detection mechanism described in Section 9.

8.6 Scalability

Although FT-TCP is engineered to minimize interference among the connections that are monitored by it, there are several data structures in the system that had to be protected by locks to avoid races. Furthermore, all messages bound for the stable buffer are serialized because they are funneled through a single kernel-level socket. To see whether these points of contention lead to significant overhead, we ran experiments involving multiple connections.

Throughput scalability. When a TCP server is handling multiple connections that perform long bulk transfers, in a steady state each connection gets a fair share of the bandwidth. When the bottleneck bandwidth BW is the same for each of n clients, every connection can achieve the same throughput of BW/n. As we kept increasing the number of incoming ttcp connections from 2 to 128 in powers of 2, as shown in Table VIII, the throughput reached by each connection was, indeed, being cut in half. Although connections under FT-TCP saw an increase (sometimes a significant one) in average packet latency, that increase did not have affect the throughput reached by each connection.

Because it is impossible to start multiple connections at the exact same instant, the earliest connections would run faster at first, but eventually would slow down. Similarly, as some connections would finish, the later ones would see a boost in bandwidth. To obtain meaningful measurements, the throughput was calculated for a 5-s stable period in the middle of the transfer.

DSS and Apache. When we measured the throughput of DSS and Apache with a single connection, we found no significant overhead imposed by FT-TCP. This is not surprising. For one thing, both applications primarily *send* data, so the load on the stable buffer is small. Unlike in Samba, responses in HTTP do not require intermittent application-level acknowledgments, so the sequence of send calls in Apache is never interrupted by other system calls. As for DSS, it throttles its outgoing streams to a relatively small bandwidth appropriate for streaming multimedia content, so FT-TCP has no problems keeping up with it.

More interesting results came out of running a standard benchmark on the Apache server and increasing both the number of clients and the number of

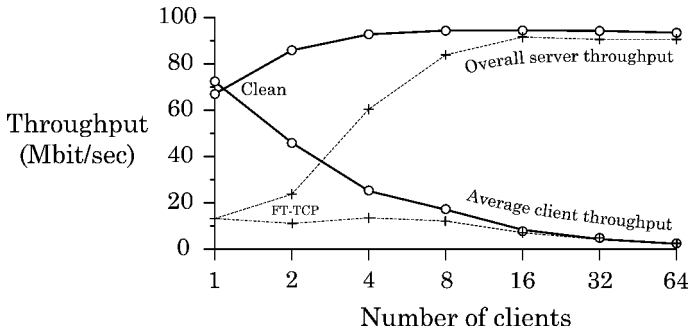

Fig. 6. Overall and per-client average throughput, with FT-TCP (dashed lines) and without (solid lines), as reported by the Webstone benchmark test of the Apache Web server with the number of clients increasing from 1 to 64, in powers of 2.

simultaneous connections. The *Webstone benchmark* is a popular tool for coordinating multiple hosts to request simultaneously static content from a Web server. We used Webstone version 2.5 (without CGI) with Apache 1.3.27 in the multiprocess configuration. We configured the benchmark to run from 1 to 64 clients in powers of 2; we used 8 physical machines, so beyond 8 clients each machine hosted multiple client applications. After a warm-up period, which ensured that the entire 5.5MB test file set, consisting of 5 files varying in size between 500 bytes and 5MB, was cached in memory, each test ran for 1 m. *Webstone* collects results from each of the clients and computes average values for various performance metrics, two of which we present in Figure 6: overall and per-client throughput. Since the benchmark does not report error bounds for the averages that it computes, it is not possible to judge the statistical significance of small differences among measurements, but we can comment on the general trend with certainty.

The overall pattern is that with few clients FT-TCP has a significant detrimental effect on the performance, but as the number of clients increases, the performance of Apache under FT-TCP gets close to (within a few percent points of) Clean performance. This counterintuitive result is the consequence of the latency overhead of FT-TCP with small requests, as explained in Section 8.5. When relatively few clients are interacting with the server, the latencies add up, resulting in large overhead (e.g., throughput of only 18% of Clean in the worst case); however, with more than 16 clients, most of the latency due to replication is "absorbed" into the latency of the Web server and TCP connection management, which happen in parallel with FT-TCP buffering.

9. FAILURE AND RECOVERY

In this section, we discuss how one can minimize *failover time*, which is the length of the period during which a client's data stream is stalled. For FT-TCP the failover time is affected by the time it takes to detect the failure (the *failure detection latency*), bring the backup into the state where it can take over the connection (the *promotion latency*), and restart the flow of data on the connection (the *retransmission gap*, explained next).

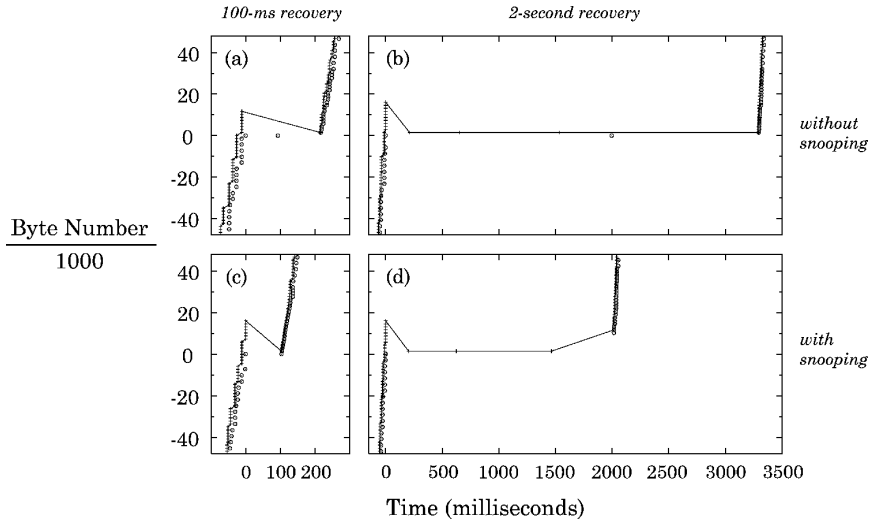

Fig. 7. Recovery scenarios for FT-TCP. The x-axis shows time, zeroed on the moment of failure; y-axis shows sequence numbers, divided by 1000 and also zeroed on the point in the stream where the failure took place. Plus symbols show client packets; circles show server's acknowledgments. The four scenarios differ by the time it takes to recover the server process and by whether snooping is used to capture the first retransmission.

In our first study [Alvisi et al. 2001] we measured the promotion latency for a cold backup as approximately 20 ms per megabyte of buffered data. With enough data in the stable buffer the promotion latency will dominate the overall failover time. We found the promotion latency of a hot backup to be considerably shorter. Hence, hot backup failover time is dominated by the failure detection latency and the retransmission gap.

9.1 Retransmission Gap

Consider the following example. Once the backup process is done rolling forward, it is fully ready to participate in the client connection; if it were to send something, that data would pass through the TCP/IP stack and emerge as a packet destined for the client. Imagine, though, that the primary was in the process of receiving data when it crashed. Upon recovery, the backup would then expect client to send it some data. Sometimes it takes the client a while to resume sending data. We call this phenomenon the *retransmission gap*.

Figure 7 contains plots of packets in four different TCP connections around the time of their failover. The x-axis shows time in milliseconds, with 0 set at the exact moment of failure. The y-axis shows packet sequence numbers, which were divided by 1000 (for readability) and shifted so that 0 is the last byte that was acknowledged by the primary before the failure. Plus symbols, which are connected with solid lines, show packets sent by the client and circles show the acknowledgment packets. In all four cases, at time 0 the primary host fails and acknowledgments from the server cease, which soon causes the client to also stop transmission of data when its TCP window fills up. We configured the

experiments so that in (a) and (c) it took about 100 ms for the backup to recover and in (b) and (d) it took around 2 s.

As soon as the server process recovers, the SSW sends an acknowledgment packet to the client, visible as a lonely circle in both (a) and (b). However, in neither case does that ACK succeed in reviving the flow of data because it acknowledged an older packet that the client's TCP already considers acknowledged. Since no new information is conveyed by that ACK, the client's TCP does nothing. It only acts when its own retransmission timer goes off: In (a) it takes place 100 ms later at about 200 ms, but in (b) it takes over a second, at about 3400 ms. The problem with (b) is that the client TCP entered the exponential back-off mode, with each retransmission taking twice as long as the previous one. The first retransmission after recovery succeeds in reviving the flow of data in both cases because as soon the backup receives that packet and acknowledges it, the clients starts sending more.

We call this time between the actual time of recovery and the time when the flow of data revives the retransmission gap. Its length depends on exactly where in the retransmission cycle recovery takes place: It can be short if the next retransmission follows soon after recovery, but it can also be long (up to 64 seconds of maximum TCP retransmission period) if the service recovers right after a retransmission. It is impossible to avoid this gap if packets arriving immediately after the crash are lost. In fact, a backup that can detect a failure and recover well under the 200 ms may inevitably have to wait that long for the first retransmission to restart the flow of data, as in case (a). Sending multiple identical acknowledgments may help because it frequently causes the client to perform a *fast retransmit*, but it doesn't always work because the fast retransmit algorithm needs a gap in sequence numbers. This effectively places a 200 ms lower limit, for both hot and cold replication, on the guaranteed failover time.[3]

9.2 Snooping

The only way to eliminate the retransmission gap is to ensure that the backup receives all the packets sent by the client. This can be done by switching the backup's network card into promiscuous mode at the beginning of the connection and snooping packets off the network shared by the client and the replicas (assuming the network equipment allows such a configuration). When the backup decides that the primary failed, it can process the snooped packets, acknowledge them, and thereby restart the flow of data immediately, as shown in (c) and (d) of Figure 7.

With this method, the failover time is limited only by the failure detection delay. From Table VII we can see that the average RTT for messages between the replicas is about 0.5 ms (although it can be shorter for smaller messages). So an adequate value for a failure detection timeout may be 1–2 ms. However, our FT-TCP implementation relies on Linux kernel timers, which in version 2.4 have granularity of 10 ms, and hence the minimal failure-detection latency and, consequently, the minimal failover time for our hot backup are 10 ms.

[3]This value is specific to recent versions of Linux and on other implementations it may be as high as 1 s [Sarolahti and Kuznetsov 2002; Paxson and Allman 2000].

Although snooping helps ensure the fastest possible failover time, looking at every packet on a busy network may place too heavy a load on the backup machine. Therefore it is worthwhile to consider a third approach, in which the network card operates normally during failure-free operation, but goes into promiscuous mode whenever a failure is detected. We call this *reactive snooping*; the first two schemes are *no snooping* and *permanent snooping*. Reactive snooping makes sense when the failure detection latency is shorter than the TCP retransmission delay (200 ms), but the promotion latency is longer. Starting to snoop before the first retransmission allows the backup to collect all packets lost in the crash and restart the data flow as soon as the promotion is complete, as, for example, happens around 2000 ms in (d). There is no point in reactive snooping with a backup that is promoted quickly since it will get the first retransmissions itself. With short promotion latency the question is whether to snoop permanently or not at all, and this is a trade-off between good failure-free performance (which would be affected by snooping) and short failover time.

The idea of using snooping to improve reliability at a low cost has been around for a long time [Powell and Presotto 1983]. Dolev et al. [1994] have used it for primary-backup replication of a network file system service. Two fault-tolerant TCP systems [Marwah et al. 2003; Orgiyan and Fetzer 2002] also rely on permanent snooping to obtain client packets on the backup.

10. RELATED WORK

One can classify solutions to the problem of connection failover according to the level at which server failures are masked. With *application-level recovery*, failures are masked from the user by the client application that attempts to reestablish broken connections. An FTP client that automatically restarts aborted transfers is an example of such recovery. NFS and Samba also rely on this recovery technique, because often their clients can recover from short disconnections transparently. Since clients need to be explicitly designed to support application-level recovery, this technique does not apply to applications that are already deployed.

Several projects have explored the idea of *socket-level recovery*, where the failure is hidden from the client by some lower layer that reestablishes connections when necessary and provides a reliable socket to the application. One such system [Snoeren et al. 2001] extends the TCP protocol with an option that enables migration of connections from one host to another. Among other things, this allows the service provider to ask the client TCP stack to migrate a failed connection to a backup. Another system that can use migration to tolerate failures is MTCP by Sultan et al. [2002], which builds upon earlier work [Srinivasan 2001; Sultan et al. 2001] from the same authors. MTCP is fine-grained in that it can migrate individual connections (not just whole processes), but it does require the server application to participate in the transfer of application state.

The system by Nasika and Dasgupta [2000] enables transparent reconnection in Windows NT without changing the TCP stack by wrapping the socket standard library routines. This system was designed to support process migration,

but can be used for fault tolerance as well. Orgiyan and Fetzer [2002] applied a similar wrapping technique to the standard C library on Linux to mask server failures. A Java-based socket-level failover mechanism has been developed by Ekwall et al. [2002]. "Rocks" is another system based on wrapping [Zandy and Miller 2002], although its goals were to mask connection failures due to network problems rather than server crashes. This last paper describes two approaches to connection recovery, one of which relies on the interception of packets, just like our system. The main drawback of socket-level recovery is that it requires upgrading some of the infrastructure (operating system, protocol stack, or middleware) on the client host.

Server-side recovery restricts the fault-tolerance logic to the server cluster. This is the easiest solution to deploy: As soon as the servers are fault tolerant, then any client can benefit from greater reliability. Our first work [Alvisi et al. 2001] demonstrated the feasibility of efficient server-side recovery and the follow-up [Zagorodnov et al. 2003] expanded on that by evaluating our approach with two well-known replication techniques and for two real-life applications.

Two similar systems were presented at the same conference as our second paper: *Failover* TCP [Koch et al. 2003] also replicates the connection at the server end, but instead of storing the incoming packets in a stable buffer and feeding the data directly to the NSW it injects identical packets into the backup's TCP layer. This implies that for a purely deterministic application no NSW is necessary, which may carry some performance advantages. ST-TCP [Marwah et al. 2005, 2003], building on earlier work by the same authors [Fetzer and Mishra 1999; Orgiyan and Fetzer 2002], is a special version of the TCP stack that enforces identical connection state, such as the choice for the initial sequence number, on the replicas and does not discard TCP buffer data on the primary until both the client and the backup have acknowledged it. Notably, ST-TCP uses snooping (called "tapping" in the paper) to obtain client packets on the backups during failure-free execution without involvement of the primary. Such technique could be used with FT-TCP as well.

Server-side recovery techniques differ mostly in their approach to maintaining consistency of applications and transport-layer drivers on replicas. Some systems assume that the application is deterministic with respect to network input [Alvisi et al. 2001; Koch et al. 2003; Marwah et al. 2005; Zhang et al. 2004], and therefore make no attempt to synchronize application state. To support more applications, one can either modify the application to make it deterministic, as we did with Samba and DSS, or ensure that its inputs (and, for multithreaded applications, their execution paths) are the same on all replicas. The former approach is application-specific and potentially laborious, although with proper OS support, such as the Continuation Box abstraction [Sultan et al. 2003], it can be made easier.

Virtualizing the entire process environment (or the entire machine) allows any service to be replicated without source code changes, but adds a steep performance penalty. For example, Hypervisor [Bressoud and Schneider 1996] had roughly a factor of 2 overhead in execution and TFT [Bressoud 1998] reported between 23% and 58% overhead for gzip, depending on the compression level. The late resurgence of virtualization technologies has, in turn, revitalized research

in process migration and failover mechanisms at the granularity of Virtual Machines (VMs). For instance, the original support for live migration of Xen VMs [Clark et al. 2005] has been extended to work across a WAN by Bradford et al. [2007] and to implement transparent VM failover in Remus [Cully et al. 2008]. All three of these mechanisms preserve ongoing TCP connections across the migration or, in the case of Remus, across the failover. The state of the TCP stack is synchronized as part of the overall synchronization of all in-memory and on-disk state of the VMs involved in the migration or failover (unlike the explicit TCP state synchronization in FT-TCP). Remus performs state transfers from primary to a backup periodically and delays outgoing network packets in output commit stalls. This scheme also imposes significant overhead because of the large amount of state being synchronized, but it solves the problem of nondeterminism in the most general case. Finding the right trade-off is an active area of research [Slember and Narasimhan 2004; Zagorodnov and Marzullo 2005].

Transport-layer state can be synchronized with or without modifications to the OS networking code on the server. Failover TCP [Koch et al. 2003] and FT-TCP do not require such changes and instead modify incoming and outgoing packets to compensate for transport-layer state differences. ST-TCP [Marwah et al. 2005, 2003] and CoRAL [Aghdaie and Tamir 2003, 2002] do. A high-level overview of another system using a custom TCP stack was presented at LISA'02 [Burton-Krahn 2002].

Backdoors [Sultan et al. 2005] is a novel approach to reducing the state synchronization latency. Instead of exchanging state during failure-free execution, they use "Intelligent" Network Interface Cards (I-NICs) with Remote Direct Memory Access (RDMA) capability, such as Myrinet cards, to extract the state necessary for recovery from a failed machine. This technique works even with a machine that has suffered an OS crash, although not with a hardware crash that causes loss of power, such as overheating. Zhang et al. [2004] do not synchronize application or transport-layer state at all, and instead have the backups infer connection state based on incoming packets. Their approach, however, assumes that applications are deterministic, that there is never any packet loss within the cluster, and that a front-end packet switch is made fault tolerant using some other technique.

Several projects studied connection failover of specific servers. A content-aware distributor has been used to resubmit failed HTTP requests [Yang and Luo 2000; Luo and Yang 2001]. [Aghdaie and Tamir 2003, 2002] have developed a protocol similar to ours that is specialized to HTTP request/reply pairs. In doing so, they are able to avoid the problem of server nondeterminism. Rescorla et al. [2002] present a solution for client-transparent failover of clustered SSL accellerators. Daniel and Choi [1999] sketch out an architecture for a replicated NFS server, building on earlier work [Peyrouze and Muller 1996] in this vein.

A more ambitious TCP server-side recovery approach is described in Shenoy et al. [2000], which proposes using several router-level redirectors scattered across the Internet to fan out packets to several geographically distributed replicas. While deploying redirectors may be a costly endeavor, this system has the benefit of tolerating WAN partitions in addition to failures that are local to the server.

Finally, there are several projects that have applied the *state-machine approach* (also known as *active replication*) to TCP-based services. One work [Basile et al. 2002] that describes "triplicating" an Apache Web server is notable for developing an algorithm (which was improved in Basile et al. [2003]) for enforcing determinism in a multithreaded application; such an algorithm could be incorporated into FT-TCP. A similar algorithm for multithreaded Java applications was presented in Napper et al. [2003]. Another work [Rodrigues et al. 2001] describes an NFS server that can tolerate Byzantine failures. While such systems can tolerate a larger class of failures, the voter mechanism used in active replication imposes a performance penalty.

11. CONCLUSION

We have described the architecture and performance of FT-TCP, a software that wraps a standard TCP layer to mask server failures from unmodified clients. We have implemented a prototype of FT-TCP, applied it to three real applications, namely Samba, DSS, and Apache, and studied its performance under failure-free execution and for executions with failures. Our implementation does not change the TCP stack and can be applied to a standard, running Linux kernel. We experimented with several network setups and found the following.

—Of the two kinds of setups that we tested, symmetric and asymmetric, the system runs best on the latter one, where the link between the replicas is faster than the link to the client. For a 10-100 setup we see no significant overhead with any applications, for 100-1000 only Samba takes a performance hit of 5% on an incoming transfer and a 4% hit on the outstream.

—For real applications, using the Eager ack strategy on the client connection and turning off Nagle's algorithm on the interreplica connection yields the best results. The Lazy ack strategy should be avoided.

—Performance of a hot backup with FT-TCP is practically indistinguishable from performance of a cold backup (with no checkpoints). Given the short recovery time of a hot backup, it is clearly the better choice.

—The largest contributor to FT-TCP overhead is the logging of packets, while interception overhead and logging of system calls are secondary. This would imply that avoiding interception and logging of system calls will gain little in throughput.

—While it was necessary to modify the code of two existing services to have them be recoverable using FT-TCP, the modifications were few. For both services, the nondeterminism was explicitly introduced by the service: For Samba, nonces and file handles are generated, and for DSS, session IDs are generated. This experience implies that adding a protocol-specific "hook" might be useful for making it easier to ensure that the backup makes the same nondeterministic choices that the primary does.

—The failover time of FT-TCP can be made very short, but to do so requires the backup to capture the data sent by the client immediately before the server failed. This requires the backup to snoop on the incoming traffic by setting its network interface to promiscuous mode. For servers that have a large

promotion latency, the backup need only start snooping when it suspects that the primary has failed, while if the promotion latency is under 200 ms then the backup should start snooping as soon as it starts executing. The use of snooping, however, only enhances performance, and is not required for server-side recovery.

REFERENCES

AGHDAIE, N. AND TAMIR, Y. 2002. Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In *Proceedings of the 11th IEEE International Conference on Computer Communications and Networks (ICCCN)*, 63–68.

AGHDAIE, N. AND TAMIR, Y. 2003. Fast transparent failover for reliable web service. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*.

ALVISI, L., BRESSOUD, T. C., EL-KHASHAB, A., MARZULLO, K., AND ZAGORODNOV, D. 2001. Wrapping server-side TCP to mask connection failures. In *Proceedings of the IEEE InfoCom Conference*, 329–337.

APACHE. 2005. Apache homepage. http://www.apache.org/.

BASILE, C., KALBARCZYK, Z., AND K., I. R. 2003. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 149–158.

BASILE, C., KALBARCZYK, Z., WHISNANT, K., AND IYER, R. K. 2002. Active replication of multithreaded applications. Tech. rep. CRHC-02-01, University of Illinois.

BHIDE, A., ELNOZAHY, E., AND MORGAN, S. 1991. A highly available network file server. In *Proceedings of the USENIX Winter Technical Conference*, 199–205.

BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. 2007. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, 169–179.

BRESSOUD, T. 1998. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (SRDS)*, 128–137.

BRESSOUD, T. AND SCHNEIDER, F. 1996. Hypervisor-Based fault tolerance. *ACM Trans. Comput. Syst. 14*, 1, 80–107.

BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F., AND TOUEG, S. 1992. Primary-Backup protocols: Lower bounds and optimal implementations. In *Proceedings of the 3rd IFIP Conference on Dependable Computing for Critical Applications*, 187–198.

BURTON-KRAHN, N. 2002. HotSwap - Transparent server failover for Linux. In *Proceedings of the 16th Systems Administration Conference (LISA'02)*, 205–212.

CLARK, C., FRASER, K., H, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. 2005. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 273–286.

CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 161–174.

DANIEL, E. AND CHOI, G. S. 1999. TMR for off-the-shelf Unix systems. Short presentation at *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*.

DOLEV, D., MALKI, D., AND YAROM, Y. 1994. Warm backup using snooping. In *Proceedings of the 1st International Workshop on Services in Distributed and Networked Environments (SDNE)*, 60–65.

DSS. 2005. Homepage. http://developer.apple.com/darwin/projects/streaming/.

EKWALL, R., URBÁN, P., AND SCHIPER, A. 2002. Robust TCP connections for fault tolerant computing. In *Proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS)*, 501–508.

ELNOZAHY, E., ALVISI, L., WANG, Y., AND JOHNSON, D. 2002. A survey of rollback-recovery protocols in message passing systems. *ACM Comput. Surv. 34*, 3, 375–408.

FETZER, C. AND MISHRA, S. 1999. Transparent TCP/IP based replication. Short presentation at *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*.

HERTEL, C. 2003. *Implementing CIFS: The Common Internet File System*. Prentice Hall. http://ubiqx.org/cifs/.

JACOBSON, V. 1988. Congestion avoidance and control. *Comput. Commun. Rev. 18*, 4, 314–329.

KOCH, R. R., HORTIKAR, S., E., M. L., AND M., M.-S. P. 2003. Transparent TCP connection failover. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 383–392.

LUO, M. AND YANG, C. 2001. Constructing zero-loss web services. In *Proceedings of the IEEE InfoCom*, 1781–1790.

MARWAH, M., MISHRA, S., AND FETZER, C. 2003. TCP server fault tolerance using connection migration to a backup server. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 373–382.

MARWAH, M., MISHRA, S., AND FETZER, C. 2005. A system demonstration of ST-TCP. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 308–313.

NAGLE, J. 1984. Congestion control in IP/TCP internetworks. RFC 896, Network Working Group. January.

NAPPER, J., ALVISI, L., AND VIN, H. 2003. A fault-tolerant Java virtual machine. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 425–434.

NASIKA, R. AND DASGUPTA, P. 2000. Transparent migration of distributed communicating processes. In *Proceedings of the 13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS)*.

ORGIYAN, M. AND FETZER, C. 2002. Tapping TCP streams. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, 278–289.

PAXSON, V. AND ALLMAN, M. 2000. Computing TCP's retransmission timer. RFC 2988, Network Working Group. November.

PEYROUZE, N. AND MULLER, G. 1996. FT-NFS: An efficient fault tolerant NFS server designed for off-the-shelf workstations. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 64–73.

POWELL, M. AND PRESOTTO, D. 1983. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th Symposium on Operating Systems Principles (SOSP)*, 100–109.

RESCORLA, E., CAIN, A., AND KORVER, B. 2002. SSLACC: A clustered SSL accelerator. In *Proceedings of the 11th USENIX Security Symposium*, 229–246.

RIJSINGHANI, A. 1994. Computation of the Internet checksum via incremental update. RFC 1624, Network Working Group. May.

RODRIGUES, R., CASTRO, M., AND LISKOV, B. 2001. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 15–28.

SAROLAHTI, P. AND KUZNETSOV, A. 2002. Congestion control in Linux TCP. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 49–62.

SHENOY, G., SATAPATI, S., AND BETTATI, R. 2000. HydraNet-FT: Network support for dependable services. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, 699–706.

SLEMBER, J. G. AND NARASIMHAN, P. 2004. Using program analysis to identify and compensate for nondeterminism in fault-tolerant, replicated systems. In *Proceedings of the 23rd International Symposium Reliable Distributed Systems (SRDS)*, 251–263.

SLYE, J. AND ELNOZAHY, E. 1996. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 250–259.

SMB. 2005. Samba homepage. http://www.samba.org/.

SNOEREN, A., ANDERSEN, D., AND BALAKRISHNAN, H. 2001. Fine-Grained failover using connection migration. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 221–232.

SRINIVASAN, K. 2001. M-TCP: Transport layer support for highly available network services. M.S. thesis, Rutgers University. Available as Tech. Rep. DCS-TR-459.

SRISURESH, P. AND HOLDREGE, M. 1999. IP network address translator (NAT) terminology and considerations. RFC 2663, Network Working Group. August.

STEVENS, R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley.

SULTAN, F., BOHRA, A., GALLARD, P., NEAMTIU, I., SMALDONE, S., PAN, Y., AND IFTODE., L. 2005. Recovering internet service sessions from operating system failures. *IEEE Internet Comput. 9*, 2, 17–27. Extended version available as Rutgers University Tech. rep. DCS-TR-524.

SULTAN, F., BOHRA, A., AND IFTODE, L. 2003. Service continuations: An operating system mechanism for dynamic migration of Internet service sessions. In *Proceedings of the Symposium Reliable Distributed Systems (SRDS)*, 177–186.

SULTAN, F., SRINIVASAN, K., IYER, D., AND IFTODE, L. 2002. Migratory TCP: Connection migration for service continuity in the Internet. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 469–470.

SULTAN, F., SRINIVASAN, K., AND IFTODE, L. 2001. Transport layer support for highly-available network services. Tech. rep. DCS-TR-429, Rutgers University. May.

X/OPEN. 1992. *Protocols for X/Open PC Interworking: SMB, Version 2*. X/Open Company Ltd. Also available at http://www.opengroup.org/products/publications/catalog/c209.htm.

YANG, C. AND LUO, M. 2000. Realizing fault resilience in web-server cluster. In *Proceedings of the Supercomputing Conference*.

ZAGORODNOV, D. AND MARZULLO, K. 2005. Managing self-inflicted nondeterminism. In *Proceedings of the 1st Workshop on Hot Topics in System Dependability (HotDep)*, 323–328.

ZAGORODNOV, D., MARZULLO, K., ALVISI, L., AND BRESSOUD, T. 2003. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 393–402.

ZANDY, V. AND MILLER, B. 2002. Reliable network connections. In *Proceedings of the 8th ACM International Conference on Mobile Computing and Networking (MobiCom)*, 95–106.

ZHANG, R., ABDELZAHER, T. F., AND STANKOVIC, J. A. 2004. Efficient TCP connection failover in web server clusters. In *Proceedings of the IEEE InfoCom Conference*. Vol. 2, 1219–1228.

Received September 2005; revised April 2008; accepted April 2009