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

DMITRII ZAGORODNOV
University of Tromsø

KEITH MARZULLO
University of California, San Diego

LORENZO ALVISI
The University of Texas at Austin

THOMAS C. BRESSOUD
Denison University

August 25, 2005

Abstract

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.2.4 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*TCP/IP*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Fault-tolerant computing system, primary/backup replication, message logging, FT-TCP, connection failover

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 Border Gateway Protocol (BGP). As more people come to rely on network services, the issue of reliability has become vital. 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. They 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, it will be directed to a healthy server. Although this client-assisted recovery works well 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 can not 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. This is also the case with BGP [27]—a TCP-based protocol deployed on the border routers that connect different administrative domains of the Internet—where peers are run by different organizations. Convincing everyone to upgrade to a new routing protocol is not a trivial task.

Exposing failures to clients is also inefficient. A client may have internal state associated with the open TCP connection to the server; losing the connection then requires the client to redo a significant amount of work. For example, a client with a connection to an Oracle server will abort all open transactions if the server fails. A similar case occurs with popular Samba clients: when a server fails, all transfers are aborted and the user must explicitly restart any outstanding transactions. With BGP, the problem is more severe: a router observing a break in the connection with its peer starts a flood of messages to other BGP routers indicating that the peer has failed; upon recovery there is yet another flood to indicate that the peer is once again available. In situations where the connection break is caused by a planned software upgrade or when the failed router can recover quickly, all these messages are undesirable, as they consume resources and unnecessarily destabilize the routing tables.

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 it 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.

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 [32], Darwin Streaming Server [13], and the Apache [4] Web server. FT-TCP supports two common application-level replication methods: *primary-backup* [9] and *message-logging* [15]. In our experiments, we found their failure-free performance characteristics statistically indistinguishable. Neither one incurred significant overhead on connection throughput and their impact on latency is small enough to be unnoticeable by users of the service. We also found that with primary-backup the failover time of FT-TCP can be made very short, but to do so the backup must aggressively capture client data.

The remainder of this article is organized as follows. We describe the general structure of the system—primary-backup as well as message-logging versions—in Section 2. In Section 3 we cover the details of the protocol used within FT-TCP, while in Section 4 we discuss the three applications we used to evaluate the protocol. The performance discussion is divided in two parts: in Section 5 we discuss the overhead of FT-TCP in terms of throughput and latency, and in Section 6 we look at the dynamics of connection failover. We compare FT-TCP to other possible approaches and alternative systems in Section 7. Finally, we draw our conclusions in Section 8.

2 Architecture

In this section, we first introduce several concepts that are relevant to the discussion of service failover, including the operation of TCP and fault-tolerance fundamentals. We then describe the general structure and operation of FT-TCP.

2.1 TCP Overview

TCP implements a bi-directional byte stream by fragmenting data into segments (usually up to 1480 bytes in length) and by sending each segment in a packet with its own header. Among other things, the header tells the receiver where in the stream the data contained in the packet belongs. That is done using *sequence numbers*. 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 zero 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 with respect to all other segments regardless of the order in which they arrived. 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. When the acknowledgment number field contains a valid value the ACK flag is also set. 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*.

Another relevant field in the TCP header is the 16-bit *window size*, which is used to tell the sender how much buffer space is available on the receiver. If, for example, the advertised window is 16 KB 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 should refrain 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 everything, including 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* [9], 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* [15], 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 pre-failure 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*, respectively. 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 referred to as the *output commit* problem [15]. We henceforth refer to these forced waiting periods as *output commit stalls*. Note that, 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.

Another issue that comes up in the context of replicated processes is *nondeterministic execution*. For both hot and cold backups, the 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, we intercept system calls on all replicas and save the system

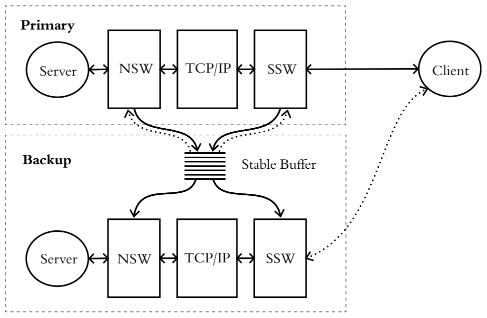

Figure 1: *FT-TCP Architecture.*

call results returned on the primary in a log, along with client requests. We discuss further how we deal with this and other sources of nondeterminism in Section 3.8.

2.3 FT-TCP Architecture

FT-TCP is implemented by "wrapping" the TCP/IP stack. By this, we mean that it can intercept, modify, and discard packets between the TCP/IP stack and the network driver using a component we call the *south-side wrap* or *SSW*. Also, FT-TCP can intercept and change the semantics of system calls (between the application and the kernel) made by the server application using a component we call the *north-side wrap* or *NSW*. Both the NSW and the SSW on the primary replica communicate with a *stable buffer* that is designed to survive crashes.

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 non-volatile 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 paper we will use a setup with a single backup and a single stable buffer, collocated with that backup, as shown in Figure 1. Note that, for cold backup, the server process on the backup machine will not be instantiated until after a failure has occurred. Such a setup is by far the most common one in practice, as running multiple backups can be exceedingly expensive, both in terms of money and performance. Nevertheless, our technique can be extended to use any number of backup hosts by slightly modifying the stable buffer protocol and by ensuring that, during failover, all backups elect the same primary.

For every open connection, the NSW and the SSW on a replica may be jointly in one of three modes:

1. 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. Fortunately, it is not also necessary to wait for the stable buffer to acknowledge packets. FT-TCP leverages the semantics of TCP, by which data must be retained at the sender until acknowledged. By acknowledging to the client only data that is stable, we have the client store segments until they are stable. The primary is in RECORD MODE until either it fails or the connection terminates normally.

2. PLAYBACK MODE: 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. Upon entering this mode, the backup spawns its own copy of the server process and provides that 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 and is used to ensure 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 quietly discarded. This mode ends either when the connection terminates normally or when a backup replica is promoted (i.e. switched into the RECORD MODE).

3. STANDBY MODE: Cold backups are in STANDBY MODE until either they are promoted or until they are reconfigured to be a hot backup. If there was no need to recover a connection during its lifetime, the replica leaves STANDBY MODE when the client connection is shut down.

As noted above, 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. Details of each of these modes will be covered in Section 3.

The state diagram for the operational modes of FT-TCP replicas is shown in Figure 2.

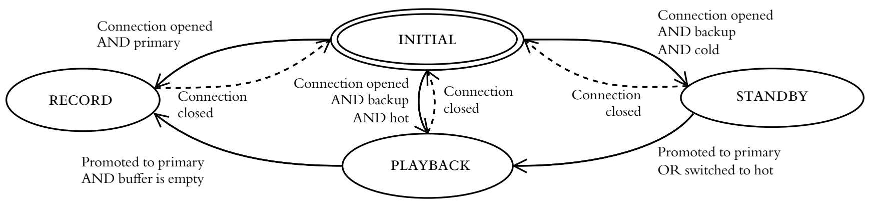
INITIAL
RECORD
STANDBY
PLAYBACK
Connection opened AND primary
Connection closed
Connection opened AND backup AND hot
Connection closed
Connection opened AND backup AND cold
Connection closed
Promoted to primary AND buffer is empty
Promoted to primary OR switched to hot

Figure 2: *FT-TCP replica modes for a connection.*

2.3.1 Overview of FT-TCP Operation

During normal operation, the primary is in RECORD MODE. In the cold-backup scheme, all other replicas are in STANDBY MODE. In the hot-backup scheme, the replicas are in PLAYBACK MODE [1].

A failure of the primary is detected by the backup(s) based on the absence of communication by the primary for an interval of time (see Section 6 for additional details on failure detection and recovery durations.) Once the failure is detected, the backup initiates failover by changing the operational mode of the replica. The details of this transition differ for the hot-backup and the cold-backup schemes.

In the hot-backup scheme, the replica continues in PLAYBACK MODE for a very short interval to consume any TCP segments or system call results remaining in the stable buffer[2]. It then may be promoted to primary and enter RECORD MODE. One of the key advantages of the hot backup approach is its short failover time, as it only requires bringing the backup process up to date by processing a few segments and system calls that the backup may have received before the primary failed before the backup can be promoted.

In the cold-backup scheme, the operational mode of the replica proceeds through two transitions. First, the replica transitions from STANDBY to PLAYBACK MODE, allowing it to *roll forward* by consuming segments and system call results from the stable buffer until it is in a state consistent with the client. Once this is achieved, the replica is promoted to primary and transitions to the RECORD MODE. Naturally, rolling the process forward takes time, hence recovery with a cold backup can be considerably slower than with a hot backup. Recovery from

[1]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.

[2]The stable buffer is typically empty by this point, as a replica in PLAYBACK MODE would have consumed any remaining buffers during the duration of the failure detection timeout interval.

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 paper—henceforth, we assume that a restarting server has the application restart from its initial state.

During failover, it is important to prevent connection timeouts and the appearance of a non-responsive 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 zero. 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.

Different techniques can be used to reconcile the difference in IP addresses of the primary host and the promoted backup. 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 [35]. To gain access to all incoming packets (that are destined to a different MAC address), we place the network interface card in 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* [7]), then using promiscuous mode may not be necessary.

3 Protocol

In this section we describe the operation of FT-TCP in detail. After introducing the state maintained by FT-TCP, we discuss how the NSW and the SSW enter and exit each of the three modes of operation (RECORD, PLAYBACK, and STANDBY) and describe the responsibilities of the wraps for each of these modes. Finally, we cover additional material including sources of nondeterminism and the details of inter-replica communication.

3.1 Variables

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

- *idelta_seq* and *odelta_seq*: The deltas (for instream and outstream, respectively) between the sequence numbers in use by the client and the sequence numbers apparent to the TCP stack at the server. 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, respectively.

- *stable_seq*: The smallest sequence number of the instream that has not yet been acknowledged by the stable buffer to the SSW. Note that this can never be larger (ignoring the 32 bit wrap) than the largest sequence number 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 variable is also used as the ID of the next system call.

- *unstable_syscalls*: The count of the number of system calls whose records have not been acknowledged by the stable buffer. If *unstable_syscalls* is zero, then the NSW knows that the stable buffer has recorded the results of all prior system calls.

3.2 Entering and Exiting Record Mode

The primary replica enters RECORD MODE as soon as the server process is initialized. Before a single instruction in the process is allowed to execute, the NSW sets *syscall_id* and *unstable_syscalls* to zero 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 server's TCP segment that acknowledges the client's SYN packet until 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 zero 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 chosen to handle the connection. As that backup completes the failover procedure, it switches from the PLAYBACK 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 3.9 .

3.3 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, respectively.

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, and 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 [28]. 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 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 re-maps 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 sufficient buffering 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.

3.4 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

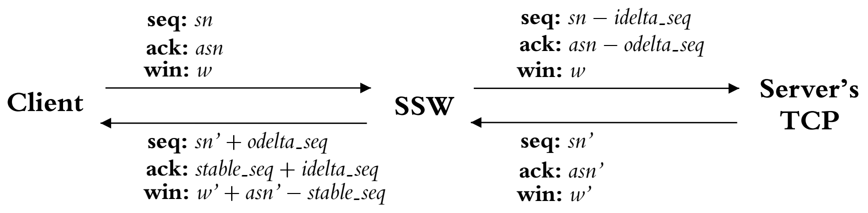

Figure 3: Record mode operation of the SSW.

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*. Note that the message content of a network read 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 network writes is not sent because backup processes will generate an identical message on their own. A short hash of the message may 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[3], the NSW blocks in an output commit stall until *unstable_syscalls* is zero.

3.5 Entering Playback Mode

A replica enters PLAYBACK MODE either at connection establishment time (for a hot backup) or as part of the failover procedure (of a cold backup). Initially, FT-TCP sets *stable_seq* and *server_seq* to the values obtained from the stable buffer, and *unstable_syscalls* is set to zero.

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 (with appropriate MAC and IP address), and has an initial sequence number of *stable_seq*. The TCP stack responds with a segment acknowledging the client SYN and with a proposed initial sequence number for the server. This segment is captured by the SSW, which sets *odelta_seq* to the difference between the original initial sequence number logged to the stable buffer and the initial sequence number from the current outgoing segment. Discarding the outgoing segment, the SSW finally creates an acknowledgment that appears to be from the client and passes it to the TCP stack.

[3]These are system calls that affect the environment.

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 $stable\_seq + 1$.

3.6 SSW in Playback Mode

We made a design decision to feed network data to the backup replica directly through the NSW (unlike the system of Koch et al. [19] that 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 is efficient, since the data takes the shortest path to its destination: from the stable buffer directly into the application's buffer. This also 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 and retransmission are done by the TCP on the primary.

3.7 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 *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. The send buffer is cleaned when client acknowledgments arrive on the backup. For debugging, the 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—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 takes care of 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 code on the backup. In particular, if a non-blocking 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.

The last category of system calls can be quite complex in both their semantics and side-effects. We consider these general cases of nondeterminism outside the scope of this paper. 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.

3.8 Nondeterminism During Playback

For replicas to execute deterministically during playback, it is not sufficient to ensure identical input. Error conditions and asynchronous events must be delivered consistently, too.

Our experience with real applications shows that capturing and replaying the value of system calls that returned the status of a socket—namely select and poll—is necessary for ensuring deterministic execution. 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, poll must return the same result on both replicas.

Two additional important cases exist. First, just like poll, a non-blocking 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. Readlengths from the primary are forwarded to the backup to ensure deterministic execution. Although we found the readlength of -1 to be a source of nondeterminism, returning the same data in chunks of different size (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 any given read, probably because they use a protocol based on messages of predetermined size.

A second important source of nondeterminism arises when several processes compete for a file lock, as 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, too. Neither proved to be problematic for the three services that we evaluated. This is not to say that a service like Samba does not use signals (in fact, we verified through code inspection that it does), but signals do not occur often enough to warrant our immediate attention. Building a commercial fault-tolerant TCP system would require capturing and replaying signals at the appropriate times in the execution path [8, 31] and implementing efficient deterministic thread scheduling [5, 22].

Finally, we had to address the nondeterminism introduced when a server generates a random value and then uses it in communications with the client. The next section 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 are considering using a protocol-specific “hook” to capture randomly generated values and make the appropriate substitutions.

3.9 Exiting Playback Mode

When a connection terminates normally, the backup replicas in PLAYBACK MODE eventually shut down—but if a failure is detected, the backup transitions from PLAYBACK MODE to RECORD MODE. In that situation, the backup first executes all the system calls that the primary has executed prior to failing. It then sets the delta values appropriately:

$$idelta\_seq = stable\_seq - isn_C - 1 \tag{1}$$

$$odelta\_seq = server\_seq - isn_S - 1 \tag{2}$$

This has the effect of translating instream and outstream 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.

3.10 Stable Buffer Protocol

In our implementation both wraps on the primary communicate with the stable buffer using a TCP connection. On the backup, the wraps 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 IDs for quickly finding the appropriate queue for that message, and some metadata such as sequence numbers and system call IDs. 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 wraps 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. Note that 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.

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. If the primary does not acknowledge the heartbeat probe within a certain time, the backup assumes that the primary has failed and initiates failover.

3.11 Ack Strategies

As described in Section 3.3, 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 an important parameter for TCP. 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 [18] 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. 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. We show in Section 5 how these strategies perform in practice.

3.12 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 [21] should be disabled for the inter-replica TCP connection. When Nagle's algorithm is disabled, every message is placed in the 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 tradeoff in Section 5.3.

4 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. Each one handles connections differently, though—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 below in the sections dedicated to the individual servers.

Another application that we used for studying the impact of FT-TCP on throughput is *ttcp*—a simple bandwidth testing tool. 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 5), ttcp can obtain over 99% of maximum theoretical throughput of TCP/IP on Ethernet configurations up to a gigabit per second.

4.1 Darwin Streaming Server

DSS is currently 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 main 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 re-open the connection and re-position 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 multi-threaded application. It has some attributes that make it less challenging. In particular, it only reads files, making the output commit problem only an issue 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 pseudo-random 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.

4.2 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) [17, 39]. 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 are different, 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 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 make sure 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.

4.3 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 non-trivial 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.

In many cases, the ability to fail over Web server connections may not be worth the potential overhead and the hardware requirements of FT-TCP. Web server connections are typically short and their failure rarely leads to a serious service disruption—in the worst case the user may have to press the "reload" button on their browser to resolve a problem. We were primarily interested in Apache for helping us understand how FT-TCP performs when many connections are created and torn down simultaneously.

5 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 re-compilation 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 exported by 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 512-KB L2 cache and 1 GB of RAM. Each machine had two on-board *Intel Pro 1000 XT* 1-Gbps Ethernet adapters that we could configure to run at speeds of 10, 100 and 1000 Mbps. By varying speeds and wiring configurations we experimented with six different network setups:

- **10 Shared**: All machines share a half-duplex 10-Mbps broadcast Ethernet segment (implemented by a single hub). In this case the client connection competes for bandwidth with the inter-replica connection. Since the instream data is going over the same medium both on its way from client to server as well as from server to stable buffer, we cannot expect to obtain more than 50% of the link's maximum bandwidth once saturation is reached.

- **10-10**, **10-100**, **100-100**, and **100-1000**: In these experimental setups, the first number specifies the bandwidth of the client-primary link and the second number specifies the bandwidth of the primary-backup link. Physically, all machines were interconnected through a 100-Mbps switch via one of their interfaces, and the two replicas employed a direction connection between their second interfaces. Changing speeds of the two adapters on the primary machine allowed us to experiment with these four setups. All links were full-duplex.

- **1000-1000**: Although we did not have a 1-Gbps switch for connecting the client to all replicas, we created this setup by connecting one interface on the primary directly to an interface on the client, and the other to an interface on the backup. No failover is possible with this configuration since there is no redundant path between the client and the backup, but it is still useful for testing throughput.

These variations cover most setups common in commercial fault-tolerant cluster systems which are the most likely setting for FT-TCP deployment. In particular, since it is common for clients to encounter a bandwidth bottleneck on the link to the service, we consider *asymmetric setups* such as 10-100 and 100-1000 the most realistic for evaluating FT-TCP performance from the point of view of a typical client.

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 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. Since our traces are obtained on the client, the latency of client-bound packets in an outgoing stream cannot be measured, but neither are they particularly interesting.

In the tables that follow we present both throughput and latency as mean values from 15 identical experiments. Included are error bounds for a 95% confidence interval. Each experiment consisted of a 4-MB transfer, except the 1000-1000 setup where 40 MB were sent. We used default TCP buffer sizes (8 KB) for all setups except 1000-1000 where the buffer was increased to 64 KB (more on this in Section 5.2). We first gathered the results of a non-wrapped 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 commonly use percentage of Clean throughput obtained by FT-TCP connections as the key measure of overhead. Finally, the tables show the average number of TCP acknowledgments sent back to the sender during a connection.

5.1 Throughput of ttcp

Table 1 shows how incoming ttcp transfer behaves on the *10 Shared* network setup under three ack strategies. Delayed and Eager strategies are similar in performance

	Throughput	Error Bound	% of Clean	Average Latency	Error Bound	Ack Count
	(KB/s)	*(KB/s)*		*(ms)*	*(ms)*	
Clean	1084.29	5.99	100.00	7.84	0.02	1439
Lazy	171.44	1.28	15.81	49.60	0.11	503
Delayed	463.17	4.30	42.72	19.03	0.06	1440
Eager	464.55	10.97	42.84	18.34	0.09	2869

Table 1: *10 Shared Performance of ttcp in (with Nagle).*

with slightly larger throughput variance in throughputs of the latter. Both obtain 43% of Clean throughput, which is not too far from the maximum of 50% that they can obtain under this setup. Lazy only gets 16% and `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. To illustrate by example, say 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, 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 only sent after the ack field has been acknowledged by the stable buffer. This pattern of behavior is indicative of a fast sender and a slow receiver (see p. 279 of [36]). Both Delayed and Eager avoid this performance-draining pattern by acking more promptly.

Lazy retains this poor performance under 10-10 setup, shown in Table 2, since it was never limited by the shared link in the first place. On the other hand, Delayed and Eager take advantage of the additional bandwidth of the inter-replica link and throughput increases to 86%. We will explain in Section 5.2 why it is difficult to do much better than this under such a *symmetric setup* where the bandwidths of client-primary and primary-backup links are identical. Note that in both Tables 1 and 2 the acknowledgment count allows us to relate ack strategies to regular TCP:

	Throughput	Error Bound	% of Clean	Average Latency	Error Bound	Ack Count
	(KB/s)	*(KB/s)*		*(ms)*	*(ms)*	
Clean	1158.26	0.14	100.00	7.37	0.00	1438
Lazy	171.11	0.02	14.77	49.88	0.02	482
Delayed	993.45	0.47	85.77	8.60	0.00	1439
Eager	995.51	0.38	85.95	8.58	0.01	2874

Table 2: *10-10 Performance of ttcp in (with Nagle).*

	Throughput	Error Bound	% of Clean	Average Latency	Error Bound	Ack Count
	(KB/s)	*(KB/s)*		*(ms)*	*(ms)*	
Clean	1158.26	0.14	100.00	7.37	0.00	1438
Lazy	1158.20	0.18	99.99	7.37	0.00	1438
Delayed	1158.19	0.18	99.99	7.37	0.00	1439
Eager	1158.34	0.12	100.01	7.37	0.00	1439

Table 3: *10-100 Performance of ttcp in (with Nagle).*

Delayed sends the same number of acks as TCP, Lazy sends about a third less and Eager sends about twice as many. Clean throughput is slightly higher under 10-10 because the link runs in full-duplex mode instead of the half-duplex mode of Table 1.

When the speed of the inter-replica link is increased to 100 Mbps FT-TCP seems to no longer impose any significant overhead on the connection, as shown in Table 3. 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. This last point deserves a note, as we saw in the previous tables that the three ack strategies can generate significantly different number of acks. When the acknowledgment from the stable buffer arrives before the server's TCP stack attempts to send that same ack, the strategies behave the same: Lazy does not merge acks and Eager does not generate extra ones because they would be redundant.

		Shared		10		100		1000	
		Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~
10	Delayed	**43**	38	**86**	77	**100**	**100**		
	Eager	**43**	38	**86**	77	**100**	**100**		
100	Delayed					80	77	**100**	**100**
	Eager					**85**	77	**100**	**100**
1000	Delayed							36	57
	Eager							56	**60**

Table 4: *Relative throughput of ttcp in.*

Incoming ttcp throughput results for all six network setups are summarized in Table 4. The table shows only the rounded-off percentages of Clean throughput (i.e. values from the third column of the preceding tables). The rows determine the speed along with the ack strategy on the client-server link, while the columns determine the speed and the use of Nagle algorithm on the primary-backup link. For example, the 10-100 results may be found in the first row, labeled “10,” and the third column, labeled “100.” Each cell contains four measurements, allowing us to identify the best parameters for each network setup. The highest numbers in each cell are set in bold. Some cells are empty either because the setup is impossible (e.g. there is no shared 100-Mbps Ethernet) or it doesn’t make much sense (e.g. client link faster than the inter-replica 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%. To be fair, 40% loss took place under an extremely challenging setup where a single client was able to saturate a 1-Gbps link with incoming data—not a common situation in most organizations. In fact, we expect the real-world network configurations to be highly 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 that is in the 1000-1000 numbers, to be discussed shortly). However, we defer discussion of Nagle’s algorithm until Section 5.3 that 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. That is not surprising since outgoing data is not sent to the stable buffer and can be sent out immediately. As described in Section 3.4, write may block waiting for system calls to become stable, but since here data is written in big chunks, this overhead is negligible.

5.2 1-Gbps 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 1-Gbps link we increased the Ethernet frame size from the standard 1500 bytes to 9000 (these are frequently called *jumbo frames* in the literature). We also configured ttcp to use the largest possible TCP buffer size of 64 KB on the receiver. Larger buffers can be used together with a *window scale* option of TCP, but our implementation did not support that option. With this buffer size and the maximum segment size of 8960, which is 9000 minus 40 to account for the TCP and IP headers, our TCP stack advertises a window of 53720 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 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. 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 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 1-Gbps 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 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. In particular, this means that the time it takes to make a copy of a packet (about 25 μsec) is absorbed by the overlap.

		Shared		10		100		1000	
		Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~
10	Delayed	21	**38**	55	**75**	61	**100**		
	Eager	34	37	55	**75**	61	**100**		
100	Delayed					15	78	15	**95**
	Eager					15	**79**	15	**95**
1000	Delayed							1	38
	Eager							2	**63**

Table 5: *Relative throughput of Samba in.*

5.3 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 4-MB (or, on a 1-Gbps link, a 40-MB) file to or from the server. The throughput percentages of incoming transfer experiments are summarized in Table 5.

Just like ttcp, Samba runs best on asymmetric setups. It reaches 100% 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% to 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 impact of Nagle's algorithm on throughput. Recall that for ttcp Nagle worked best, but with Samba it consistently leads to lower throughputs and, in fact, produces increasingly disastrous results as the speed of the client link increases—around 15% with a 100-Mbps and as little as 2% on a 1-Gbps link! By examination of tcpdump traces, we determined that the root of the problem is that Samba sends data in batches (of about 64 KB), with an acknowledgment expected after every batch. Naturally, 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 6. Be-

		Shared		10		100		1000	
		Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~	Nagle	~~Nagle~~
10	Delayed	56	**80**	42	**98**	42	**99**		
	Eager	56	**80**	42	**98**	43	**98**		
100	Delayed					7	**93**	7	**96**
	Eager					8	92	8	94
1000	Delayed							3	**71**
	Eager							3	**71**

Table 6: *Relative throughput of Samba out*

cause 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. In fact, most setups are only short by 4% or less and only 1000-1000 loses 29%. This is quite a contrast to ttcp, where no overhead was measured on the outgoing transfers because it sends data in just several sends. As with the incoming transfers, it is better to turn off Nagle's algorithm. Finally, note that 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.

5.4 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 was discussed in Sections 3.3 and 3.4, 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 the packets and readlengths (results returned by read calls do matter for most applications). So, in Figure 4 we show average throughput of an incoming Samba transfer on a 100-100 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

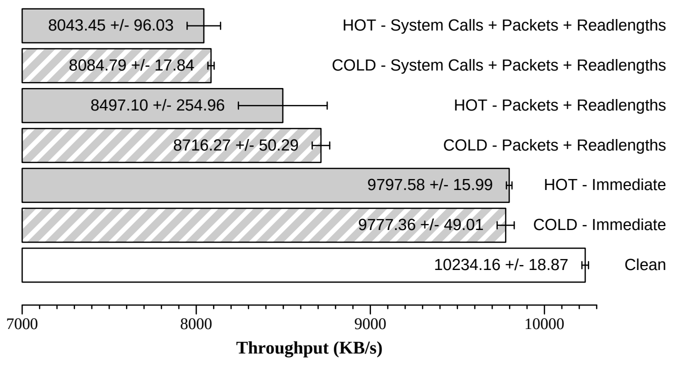

Figure 4: *Throughput of Samba in under different levels logging.*

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.

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 was 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) is from buffering of packets and readlengths. And, finally, the remaining quarter is because of 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.

Latency (ms)	Samba Request			TCP Packet			Buffering		
	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

Table 7: *Breakdown of latencies for short Samba requests*

5.5 DSS and Apache

We also looked at performance of DSS and Apache but found no significant overhead imposed by FT-TCP. For one thing, both applications primarily *send* data, so the load on the stable buffer is small. Additionally, DSS throttles its outgoing streams to some relatively small bandwidth appropriate for streaming multimedia content, so FT-TCP had no problems keeping up with it. Unlike SMB, responses in HTTP do not require intermittent acknowledgments, so the sequence of `send` calls in Apache is never interrupted by other system calls.

5.6 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 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 7 with minimum, mean, and maximum values. Also, the same mean values along with error bars for the 95% confidence interval are graphed in Figure 5. There were 30 Samba request latency measurements, 68 packet latency measurements, and 230 buffering latency measurements (which include both packet and system call requests).

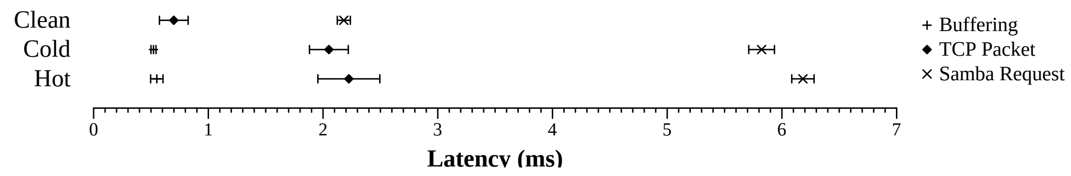

Figure 5: *Breakdown of latencies for short Samba requests.*

As the second column of the table illustrates, the average Samba request latency almost tripled—from 2.2 ms to 5.8 ms under cold and 6.2 ms under hot—when FT-TCP was added. Those values are shown as *X*s on the graph, which also reveals that the difference between cold and hot is statistically significant, since their error bars do not overlap. Although FT-TCP increases the latency over 280% in this experiment, the values are still low enough that from the human perspective responsiveness is not affected at all.

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 there is a clear correlation between them. While such an increase may be significant in some circumstances, the latencies are comparable to connection latencies experienced across a WAN. 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 3.

As Figure 5 shows, there is no statistically-significant difference between hot and cold average packet latencies, indicated with diamonds. The same is true for the FT-TCP buffering latencies, shown as crosses in the figure and as numbers in the last three columns of the table. Because the distribution of values in all of these experiments is not perfectly normal (three different types of packets produced a tri-modal distribution), the averages and their confidence levels are not very useful for understanding these data. Still, by noting that both minimal and maximal values of hot are higher than for cold, we can conclude that the increased buffering latency is the cause of the higher Samba request latency under hot replication.

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

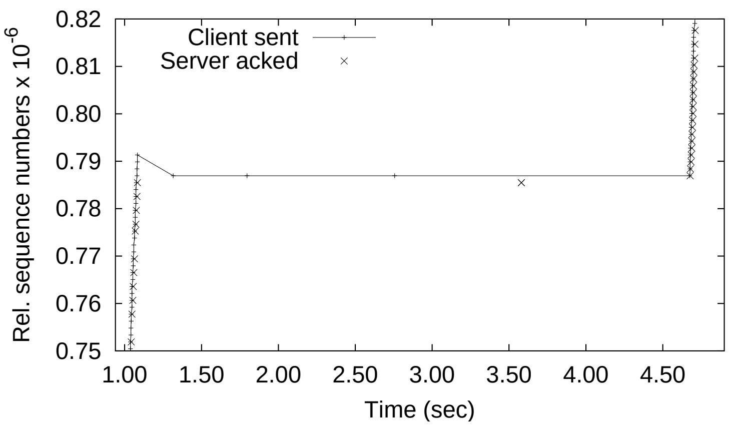

Figure 6: *Behavior of FT-TCP for a long (2.5 sec) promotion latency with no snooping.*

the next section.

6 Failure and Recovery

Here we describe how we could minimize failover time, where the *failover time* 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 (a) detect the failure (the *failure detection latency*), (b) bring the backup into the state where it can take over the connection (the *promotion latency*), and (c) restart the flow of data on the connection (the *retransmission gap*, more carefully defined below). In our earlier study [3] we measured the failover time for a cold backup—approximately 20 ms per megabyte of buffered data—and that time is dominated by the promotion latency. We found recovery of a hot backup considerably more efficient than that. Hot backup failover time is dominated by the failure detection latency and the retransmission gap. Consider the following example.

Figure 6 shows a portion of one connection by plotting sequence number offsets (relative to the beginning of the connection) of the data packets sent by the client or acknowledgment packets sent by the server. About 1 sec into the experiment the primary host is forced to fail and acknowledgments from the server cease, which soon causes the client to also stop transmission of data when its TCP window fills up. About 300 ms later the client's retransmission timer goes off and it attempts to resend the packet that follows the last acknowledged packet (shown as a dip in the line). For the purposes of analysis we forced the recovery to take 2.5 seconds and so retransmissions proceed unacknowledged at exponentially increasing intervals for three more rounds. By the fourth round, 4.8 seconds into the experiment, the backup is ready, so the retransmission succeeds and the flow of data resumes.

The actual time when the backup recovered is indicated by an ACK packet visible around 3.6 seconds. Unfortunately, that ACK does not succeed in reviving the flow of data because it acknowledges an older packet that client TCP already considers acknowledged. The length of this *retransmission gap* between the actual time of recovery and the time when the flow of data revives depends on exactly where in the retransmission cycle recovery happens to take place: it can be very short if the next retransmission follows soon after recovery, but it can also be very 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 hot backup that can detect a failure and recover well under the 200 ms will inevitably have to wait that long for the first retransmission to restart the flow of data. This effectively places a 200 ms lower limit—for both hot and cold replication—on the guaranteed failover time.

The only way to eliminate the retransmission gap is to ensure that the backup receives all of the packets sent by the client. That 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. 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 Figure 7. With this method the failover time is limited only by the failure detection delay. From Table 7 we can see that the average RTT for messages between the replicas is about 0.5 ms (although it can be much less for shorter messages). So a reasonable value for a failure detection timeout might be 1–2 ms. Unfortunately, FT-TCP implementation relies on Linux kernel timers that have granularity of 10 ms, making that the minimal failure-detection latency and consequently the minimal failover time for our hot backup.

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*, respectively. 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

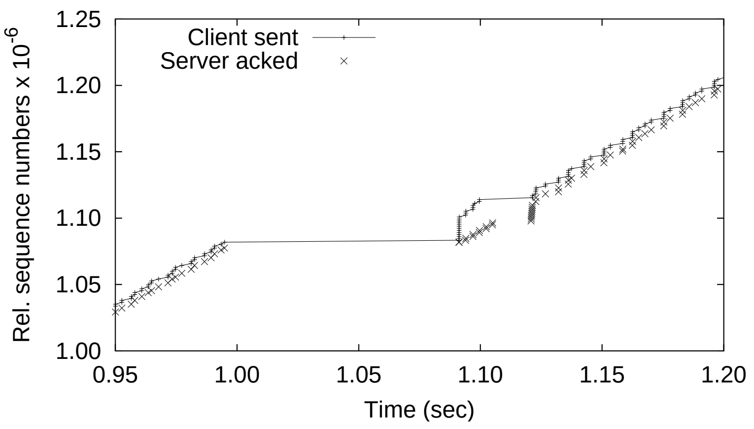

Figure 7: *Behavior of FT-TCP for a short (100 ms) promotion latency with permanent snooping.*

restart the data flow as soon as the promotion is complete, as, for example, happens around 3.4 seconds in Figure 8. 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 that 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 [26]. Dolev et al. [12] have used it for primary-backup replication of a network file system service. Two fault-tolerant TCP systems [20, 24] also rely on permanent snooping to obtain client packets on the backup.

7 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* the 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 clients also fall in this category because in many cases they can recover from short disconnections transparently. Since the client needs to be explicitly designed to support application-level recovery, this approach is inapplicable to the already deployed applications.

Several projects have explored the idea of *socket-level recovery*, where the fail-

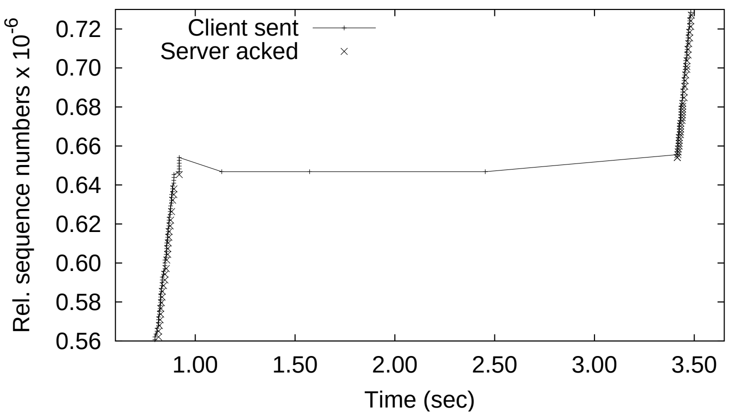

Figure 8: *Behavior of FT-TCP for a long (2.5 sec) promotion latency with permanent or reactive snooping.*

ure 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 [33] 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. [37], which builds upon earlier work [34, 38] from that team. This system 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 et al. [23] 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. ST-TCP [24] 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. [14]. "Rocks" is another system based on wrapping [42], although their 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 [3] demonstrated the feasibility of efficient server-side recovery and the followup [41] 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* [19] 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* [20], building on authors' earlier work [24, 16], is a special version of the TCP stack that enforces identical connection state, such as isn_S, 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 tapping to obtain client packets on the backups during failure-free execution without involvement of the primary. Such technique can be added to FT-TCP, as well. A high-level overview of another system using a custom TCP stack and system call interception was presented at LISA'02 [10].

Several projects studied connection failover of specific servers. A content-aware distributor has been used to resubmit failed HTTP requests [40]. The authors of [1, 2] 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. [11] sketches out an architecture for a replicated NFS server, building on earlier work [25] in this vein.

A more ambitious TCP server-side recovery approach is described in [30], 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 [6] that describes "triplicating" an Apache Web server is notable for developing an algorithm (which was improved in [5]) for enforcing determinism in a multi-threaded application—such an algorithm could be incorporated into FT-TCP. A similar algorithm for mutli-threaded Java applications was presented in [22]. Another work [29] 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 significant performance penalty.

8 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—Samba, DSS, and Apache—and studied its impact for both failure-free execution and for executions with failures. Our implementation does not change the TCP stack and can be applied on-the-fly to a standard, running Linux kernel. We experimented with several network setups and found that:

- 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 inter-replica 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 warm 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

[1] N. Aghdaie and Y. Tamir. Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In *Proc. 11th IEEE Intl. Conf. on Computer Communications and Networks (ICCCN)*, pages 63–68, Miami, Florida, USA, October 2002.

[2] N. Aghdaie and Y. Tamir. Fast transparent failover for reliable web service. In *Proc. 15th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS)*, Marina del Rey, California, USA, November 2003.

[3] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proc. IEEE INFOCOM*, pages 329–337, Anchorage, Alaska, USA, April 2001.

[4] Apache. http://www.apache.org/, 2005.

[5] C. Basile, Z. Kalbarczyk, and Iyer R. K. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 149–158, San Francisco, California, USA, June 2003.

[6] C. Basile, Z. Kalbarczyk, K. Whisnant, and R. K. Iyer. Active replication of multithreaded applications. Technical Report CRHC-02-01, University of Illinois, 2002.

[7] A. Bhide, E.N. Elnozahy, and S.P. Morgan. A highly available network file server. In *Proc. USENIX Winter Technical Conf.*, pages 199–205, Dallas, Texas, USA, January 1991.

[8] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer Systems*, 14(1):80–107, 1996.

[9] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary–backup protocols: Lower bounds and optimal implementations. In *Proc. 3rd IFIP Conf. on Dependable Computing for Critical Applications*, pages 187–198, Mondello, Italy, September 1992.

[10] N. Burton-Krahn. HotSwap - transparent server failover for Linux. In *Proc. of LISA '02: Sixteenth Systems Administration Conference*, pages 205–212, Philadelphia, Pennsylvania, USA, November 2002.

[11] Eric Daniel and Gwan S. Choi. TMR for off-the-shelf Unix systems. Short presentation at IEEE Intl. Symp. on Fault-Tolerant Computing (FTCS), June 1999.

[12] D. Dolev, D. Malki, and Y. Yarom. Warm backup using snooping. In *Proc. 1st Intl. Workshop on Services in Distributed and Networked Environments (SDNE)*, pages 60–65, Prague, Czech Republic, June 1994.

[13] DSS. http://developer.apple.com/darwin/projects/streaming/, 2005.

[14] Richard Ekwall, Péter Urbán, and André Schiper. Robust TCP connections for fault tolerant computing. In *Proc. 9th Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 501–508, Chung-li, Taiwan, December 2002.

[15] E. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[16] C. Fetzer and S. Mishra. Transparent TCP/IP based replication. Short presentation at IEEE Intl. Symp. on Fault-Tolerant Computing (FTCS), June 1999.

[17] Christopher Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall, 2003. Also available at http://ubiqx.org/cifs/.

[18] V. Jacobson. Congestion avoidance and control. *Computer Communications Review*, 18(4):314–329, 1988.

[19] R. R. Koch, S. Hortikar, Moser L. E., and Melliar-Smith P. M. Transparent TCP connection failover. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 383–392, San Francisco, California, USA, June 2003.

[20] M. Marwah, S. Mishra, and C. Fetzer. TCP server fault tolerance using connection migration to a backup server. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 373–382, San Francisco, California, USA, June 2003.

[21] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Network Working Group, January 1984.

[22] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 425–434, San Francisco, California, USA, June 2003.

[23] R. Nasika and P. Dasgupta. Transparent migration of distributed communicating processes. In *Proc. 13th ISCA Intl. Conf. on Parallel and Distributed Computing Systems (PDCS)*, Las Vegas, Nevada, USA, August 2000.

[24] M. Orgiyan and C. Fetzer. Tapping TCP streams. In *Proc. IEEE Intl. Symp. on Network Computing and Applications (NCA)*, pages 278–289, Cambridge, Massachusetts, USA, October 2002.

[25] Nadine Peyrouze and Gilles Muller. FT-NFS: an efficient fault tolerant NFS server designed for off-the-shelf workstations. In *Proc. IEEE Intl. Symp. on Fault-Tolerant Computing (FTCS)*, pages 64–73, Sendai, Japan, June 1996.

[26] M. Powell and D. Presotto. Publishing: a reliable broadcast communication mechanism. In *Proc. 9th Symp. on Operating Systems Principles (SOSP)*, pages 100–109, Bretton Woods, New Hampshire, USA, October 1983.

[27] Y. Rekhter and P. Gross. Application of the Border Gateway Protocol in the Internet. RFC 1268, Network Working Group, October 1991.

[28] A. Rijsinghani. Computation of the Internet Checksum via Incremental Update. RFC 1624, Network Working Group, May 1994.

[29] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. 18th ACM Symp. on Operating Systems Principles (SOSP)*, pages 15–28, Château Lake Louise, Banff, Alberta, Canada, October 2001.

[30] G. Shenoy, S.K. Satapati, and R. Bettati. HydraNet-FT: Network support for dependable services. In *Proc. 20th Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 699–706, Taipei, Taiwan, April 2000.

[31] J.H. Slye and E.N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proc. IEEE Intl. Symp. on Fault-Tolerant Computing (FTCS)*, pages 250–259, Sendai, Japan, June 1996.

[32] SMB. http://www.samba.org/, 2005.

[33] A.C. Snoeren, D.G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. 3rd USENIX Symp. on Internet Technologies and Systems (USITS)*, pages 221–232, San Francisco, California, USA, March 2001.

[34] K. Srinivasan. M-TCP: Transport layer support for highly available network services. Master's thesis, Rutgers University, October 2001. Available as technical report DCS-TR-459.

[35] P. Srisuresh and Holdrege M. IP network address translator (NAT) terminology and considerations. RFC 2663, Network Working Group, August 1999.

[36] R.W. Stevens. *TCP/IP illustrated, Volume 1: The protocols*. Addison-Wesley, 1994.

[37] F. Sultan, K. Srinivasan, , D. Iyer, and L. Iftode. Migratory TCP: Connection migration for service continuity in the Internet. In *Proc. of the IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 469–470, Vienna, Austria, July 2002.

[38] F. Sultan, K. Srinivasan, and L. Iftode. Transport layer support for highly-available network services. Technical Report DCS-TR-429, Rutgers University, May 2001.

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

[40] C. Yang and M. Luo. Realizing fault resilience in web-server cluster. In *Proc. Supercomputing*, Dallas, Texas, USA, November 2000.

[41] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 393–402, San Francisco, California, USA, June 2003.

[42] V.C. Zandy and B.P. Miller. Reliable network connections. In *Proc. 8th ACM Intl. Conf. on Mobile Computing and Networking (MobiCom)*, pages 95–106, Atlanta, Georgia, USA, September 2002.