Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP *

Dmitrii Zagorodnov [†] Keith Marzullo [†] Lorenzo Alvisi [‡] Thomas C. Bressoud [§]

University of California, San Diego [†]
Computer Science & Engineering
9500 Gilman Dr (MC 0114)
La Jolla, CA 92037 USA
{dzagorod,marzullo}@cs.ucsd.edu

The University of Texas at Austin [‡]
Department of Computer Science
1 University Station C0500
Austin, TX 78712 USA
lorenzo@cs.utexas.edu

Denison University [§]
225a Olin Hall
Granville, OH 43023 USA
bressoud@denison.edu

Abstract

In a recent paper [2] we have proposed FT-TCP: an architecture that allows a replicated service to survive crashes without breaking its TCP connections. FT-TCP is attractive in principle because it does not require modifications to the TCP protocol and does not affect any of the software running on the clients; however, its practicality for real-world applications remains to be proven. In this paper, we report on our experience in engineering FT-TCP for two such applications—the Samba file server and a multimedia streaming server from Apple. We compare two implementations of FT-TCP, one based on primary-backup and another based on message logging, focusing on scalability, failover time, and application transparency. Our experiments suggest that FT-TCP is a practicable approach for replicating TCP/IP-based services that incurs low overhead on throughput, scales well as the number of clients increases, and allows recovery of the service in near-optimal time.

1 Introduction

Consider a company that provides a TCP-based service on a large intranet or the Internet. The service is important; if it fails, then it needs to be restarted in a timely manner. There are some constraints the company might face when deciding how to provide service failover:

- If the client base is large and diverse, then the most important constraint can be the lack of control over the client host configuration and the applications running on the host. This means that client applications can not be expected to help in the failover of the service.
- While service outages can have a large impact on a company's business, service outages are rare. Hence, the failover approach should incur a low cost in terms of performance of the server during periods of time when there are no failures.
- Deploying the failover approach should have a low impact on the design and installation of the service, since changes in server platforms, upgrading failover software, and deploying new services can be costly. Since this is an offline cost, though, it can be less important than the first two constraints.
- Depending on the service, failover time might need to be rapid. For example, a client playing a QuickTime movie would experience visualization problems if the failover lasts too long.

Many companies marketing high-end server hardware these days—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 coordination, and for failure detection. Clients notice

*Alvisi was supported in part by the National Science Foundation (CAREER award CCR-9734185), an Alfred P. Sloan Fellowship, a grant from the Texas Advanced Technology Program, and the AFRL/Cornell Information Assurance Institute. Marzullo and Zagorodnov were supported in part by the AFOSR MURI grant 411316865 and by the DARPA grant N66001-01-1-8933. This paper appears in the **Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)**.

when a server they are connected to fails, but if they were to open another connection they would reach a healthy server.

It is often desirable to hide server failures from the clients. One reason is that the client may have state associated with the open TCP connection to the server; losing the connection may require 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 over. A similar case occurs with existing Samba clients: when a Samba server fails, all transfers are aborted and the user must explicitly restart the transaction.

We have previously shown that it is possible to hide server failures from clients [2]. We did this by building FT-TCP, which is a failover service based on message logging [6]. We evaluated the performance of FT-TCP for a synthetic application consisting of a single client having one connection open to the server. In this paper, we argue that FT-TCP can be made practicable. We show that:

- While it was necessary to modify the code of two existing services to have them be recoverable using FT-TCP, the modifications were few.
- The failure-free overhead of FT-TCP is very low and the system does not have inherent scalability problems.
- A primary-backup [4] version of FT-TCP performs almost indistinguishably from a message logging version and has the benefit of a shorter failover time.
- 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.

We compare FT-TCP to other possible approaches and already existing systems in Section 2. Architecture of the system—primary-backup as well as message-logging versions—is presented in Section 3. Our experience with getting two popular applications to run under FT-TCP is described in Section 4. Performance during normal operation, as well as behavior of FT-TCP during failover are presented in Section 5. Conclusions are drawn in Section 6.

2 Related Work

One can categorize 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 failure is hidden from the client by some lower layer that reestablishes connections when necessary and provides a reliable socket to the application. One such system [11] 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. A similar approach was adopted by [12], but it requires the server application to be aware of the replication.

The system described in [7] 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. The system described in [8] applied a similar wrapping technique to the standard C library on Linux to mask server failures. They evaluated the feasibility of having the backup snoop on packets sent to the primary and concluded that such a system would not have significant overhead. Another system based on wrapping is described in [13], 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 earlier work [2] demonstrated the feasibility of efficient server-side recovery. This work expands on that by evaluating our approach with two well-known replication techniques and for two real-life applications.

The authors of [1] share our philosophy of server-side recovery. They 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. Another TCP server-side recovery approach is described in [10], 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.

3 Architecture

In this section, we first introduce several concepts that are relevant to discussion of server-side recovery. We then describe the structure and operation of FT-TCP.

3.1 Replication Concepts

To enable recovery of a network service every connection must be backed by a number of server replicas: a primary server and at least one backup. Should the primary fail, the backups must have all the information needed to take over the connection. Several general approaches to coordinating replicas have been considered by the research community; FT-TCP supports two of them.

In the first approach, called *primary-backup* [4], every replica processes client requests and when everyone is done then one of them (the primary) replies. If the primary fails, one of the backup replicas is chosen as the new primary. In the second approach, called *message logging* [6], only one replica is active at a time and all requests from the client are saved in a log that can survive failures. Just like in the first approach, the primary does not reply until it knows that all prior requests have been logged. If a failure occurs, another replica replays messages from the log to bring it to the pre-failure state of the primary.

To make the terms more consistent, 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 knows that the backup could be recovered to the primary's current state. This is commonly referred to as *output commit* problem [6]. 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. 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, a backup may never reach the state of the primary and therefore will not be able to take over the connection.

We discuss how we deal with sources of nondeterminism that cause execution path diversions in the next two sections.

3.2 FT-TCP System

FT-TCP is implemented by "wrapping" the TCP/IP stack. By this, we mean that it can intercept, modify, and discard packets on their way in and out of the TCP/IP stack using a component we call the *south-side wrap* or *SSW*. FT-TCP can also intercept and change the semantics of system calls made by the server application using a component we call the *north-side wrap* or *NSW*. Both wraps on the primary replica place some data into a *stable buffer* that is designed to survive crashes.

In our case, the buffer is located in physical memory of the backup machines, but other approaches—such as saving data on disk or in non-volatile memory—are possible. In addition to saving data, a stable buffer can acknowledge

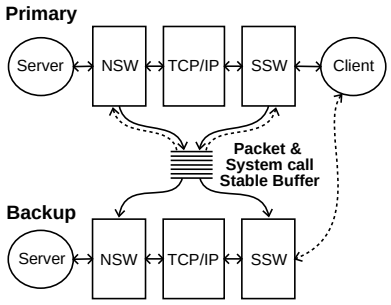

Figure 1. *FT-TCP Architecture*

the pieces of data it receives, as well as send them back in FIFO order. In the rest of the paper we will use a setup with one backup (and therefore one stable buffer, located on that backup), but our technique can be extended in a straightforward way to use any number of backup hosts. The setup is shown in Figure 1.

3.2.1 Normal Operation

During normal operation the SSW sends incoming packets to the backup and the NSW does the same with results of system calls (*syscalls*) that the application makes. Every attempt to send data to the client is suspended until all syscalls have been acknowledged. Fortunately, it is not necessary to wait for backup to acknowledge packets. Because TCP buffers outgoing data on the sender until the receiver acknowledges it, we can effectively have client store packets until the backup has them by never acknowledging to the client more than was acknowledged by the backup. Our earlier paper [2] discusses in detail how FT-TCP manipulates TCP sequence numbers to achieve this. Any packets lost in a failure will be resent through the standard TCP retransmission mechanism.

With a cold backup, nothing besides saving incoming requests in the stable buffer and acknowledging them is happening on the backup host. A hot backup, on the other hand, runs its own copy of the server process and provides that process with data that it removes from the stable buffer. To establish a connection, FT-TCP on the backup removes a buffered SYN packet, changes the destination address on that packet to its own address and injects it into its TCP stack. The stack replies with a SYN+ACK packet, which is caught and acknowledged with an ACK packet by the SSW. This 3-way handshake is not visible outside the backup host, but its TCP thinks it just received a connection from the client.

Up at the application level, the call to `accept()` returns and the server process (on both replicas) proceeds to service the incoming connection. As the process on the primary host makes syscalls, their results are sent to the backup. When a backup process makes a syscall, FT-TCP uses the corresponding syscall record from the primary to do one of several things:

- For calls that query the environment – such as `gettimeofday()` and `getpid()` – the backup immediately returns the result that the primary got;
- For a `send()`, the backup ignores the actual data passed by the server application and simply returns the result that the primary got. For debugging, the buffers 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 diversion 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 efficiently pass client data 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 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 that error on the backup even if packets with new data have arrived by the time this syscall is invoked on the backup. As we'll show in Section 3.3, this is an important source of nondeterminism.

The last category of syscalls can be quite complex in their semantics and side-effects, which is why we consider dealing with them a separate problem outside the scope of this paper. For now we just allow them to execute on the backup and assume that they will have the same effect and will return the same results as the calls on the primary. This assumption turned out to be valid for the two applications that we considered.

3.2.2 Recovery

When the backup does not hear from the primary for a certain amount of time it assumes that the primary has crashed and initiates failover (also see Section 5.4 for more details of failure detection and recovery). One of the key advantages of the hot backup approach is speed; it only requires bringing the backup process up to speed by processing any packets and syscalls that the backup received before the primary failed, and then promoting the backup to be the primary.

A number of techniques can be used to reconcile the difference in IP addresses of the primary host and the promoted backup. In the current 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 NAT. To be able to see the incoming packets (that are destined to a different MAC address), we place the network interface card into promiscuous mode. If some technique for permanently changing the IP address of the entire host is used then using promiscuous mode is not necessary.

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

Recovery with a cold backup essentially consists of redoing the actions performed by a hot backup during normal operation followed by the actions performed by it during failover. First, a new server process is started on the backup host and a connection to it is spoofed by the SSW. That process then consumes buffered packets and syscalls, and eventually takes over the connection after the IP address and sequence number adjustments that were described in the paragraph above. Naturally, processing the buffered data takes time, hence recovery with a cold backup is considerably slower than with a hot backup.

3.3 Nondeterminism

To keep replicas running deterministically, it is not sufficient to give applications identical input. Error conditions and asynchronous events must be delivered consistently, too.

In our earlier work [2] we considered the number of bytes returned by a `read()` (which we called a *readlength*) as a potential source of nondeterminism in real applications. Indeed, one can imagine an application that would perform a different action based on how many bytes were returned by a `read()`. And yet, when we purposefully returned different number of bytes on the primary and the backup, the two replicas behaved identically under both DSS and Samba. We believe that the reason is that both services either only process messages of a known size—either proto-

col header messages or data-carrying messages whose size is known from a preceding header message—or process a stream of data until it is drained. If a server processes messages of a known size and receives less than the expected number of bytes then it waits until more bytes are available—it does not process the message until it is complete.

Although applications tend to behave the same way while there are data to be received, they typically switch to a different task when no data are available. Our experience with DSS and Samba showed that capturing and replaying the value of syscalls that returned the status of a socket—namely `select()` and `poll()`—was necessary for ensuring deterministic execution. If, for example, a `poll()` on the primary indicated that there were data to be read, then it would go ahead and read those data; but if at the same point the backup was told by `poll()` that were no data, it may yield the CPU to a different thread, leading the backup process down a different execution path. Therefore, `poll()` must return the same result on both replicas.

There are two other cases. Just like `poll()`, a non-blocking `read()` has the ability to indicate the lack of data in a socket buffer (by returning -1 with `errno` set to `EAGAIN`), and that is why we consider readlengths of -1 as a source of nondeterminism. When a number of processes compete for a file lock, there is a good chance they won't all acquire it in the same order on the primary and on the backup. This means there will 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.

Thread scheduling and signal handling are both commonly identified as sources of nondeterminism, too. Neither proved to be problematic for the two services that we evaluated. That is not to say that a service like Samba does not use signals (in fact, we know that it does from looking at its source code), but that they do not occur often enough to warrant immediate attention. Someone building a commercial fault-tolerant TCP system would certainly have to capture and replay signals at the appropriate times in the execution path using a technique similar to the one used by the Hypervisor[3].

One source of nondeterminism we had to address was introduced by the servers themselves. This happens when a server generates a random value and then uses that value in communications with the client. In the next section we will show how we modified the server applications to ensure that identical random values are generated on the primary and on the backup. In the future, to avoid source code modifications we are considering using a protocol-specific "hook" to capture randomly generated values and make the appropriate substitutions.

4 Applications

To see if FT-TCP could be used to replicate non-trivial applications easily, we tested our system with two complex and well-known TCP/IP services.

We chose the *Darwin Streaming Server* (DSS) that serves multimedia content such as QuickTime movies and the *Samba* server that implements Microsoft's file and printer sharing protocols. These two services differ in their general structure—for example, Samba spawns a separate process for each client connection while DSS handles all connections in a single thread. We discuss such structural details below in the two sections on the individual servers. Besides their popularity, these applications were attractive because they tend to have long-lived connections (which are worth recovering) and their source code was publicly available.

We don't wish to imply that by having run these two services under FT-TCP, we have a complete understanding of the impact of service structure on our approach. Both DSS and Samba are relatively simple services. It does show, though, that the approach can be used at least for *some* realistic services.

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 doing 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 at least two connections: one for control of a 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 "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. NSW detected a nondeterministic diversion 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 further communication in a session. If the primary and the backup generate different IDs, then all requests from the client 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 syscalls 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

Samba server implements Microsoft's family of protocols for sharing files and printers, such as SMB and the newer CIFS. These protocols were originally designed to run over LAN transport protocols, but these days they use TCP/IP almost exclusively.

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 that we are familiar with 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. Obviously, 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.

5 Performance

FT-TCP is implemented as a kernel module for version 2.2.19 of Linux. We ran it on two identical 266MHz Pentium II workstations with 512Kb cache and 256Mb of RAM, while all clients ran under Linux 2.4.18 on a 2GHz Pentium IV with 512Kb cache and 1Gb of RAM. The client host was connected to the servers with a 10Mbps half-duplex broadcast Ethernet segment, and the servers also had a separate 100Mbps half-duplex Ethernet link between them. This basic architecture is used by many commercial fault-tolerant cluster systems and is therefore the most likely setting for FT-TCP deployment. All network interface cards were *Intel EtherExpress 10/100*. Since it is common for clients to encounter a bandwidth bottleneck on the link to the service, we consider our setup adequate for evaluation of FT-TCP performance from the point of view of a typical client.

In the next section we present results obtained during experiments without any failures and then discuss failure and recovery process in Section 5.4.

5.1 Failure-free Operation

As was discussed in Section 3.2.1, during failure-free operation FT-TCP on the primary buffers incoming packets and syscall results (readlengths are treated separately from other syscalls in the measurements that follow). To determine exactly how much overhead is introduced by interception of syscalls, by buffering of packets, and by output commit stalls, we ran FT-TCP in three different modes:

- **Immediate** - Packets and readlengths are buffered, 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.

- **PR** - (P)acket and (R)eadlengths are buffered and output commit stalls take place, adding to the overhead. If an application can run deterministically without interception of syscalls then PR mode is sufficient for correct operation.

- **PRS** - (P)ackets, (R)eadlengths and (S)yscalls are buffered. This is the full-fledged mode of FT-TCP operation that allows replication of arbitrary programs. By comparing these results with PR we were hoping to infer the additional overhead of intercepting syscalls and stalling on output commit on their behalf.

All three modes are compared to performance of the service without FT-TCP—labeled *Clean TCP*—which is the optimal performance in our case. Throughput values were computed by timing large (4Mb) data transfers and averaging the results over 20 runs. Care was taken to ensure that each run started in the same initial state (i.e. a file transfer started with a cold disk cache). Error bars in graphs represent confidence limits for the mean for a 95% confidence interval.

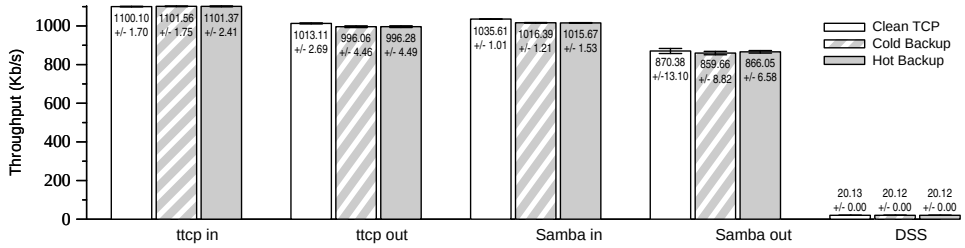

Figure 2. *Throughput of ttcp, Samba, and DSS with and without FT-TCP*

5.2 Throughput

Throughput measurements for all applications that we considered are shown in Figure 2. Because of protocol overhead in Samba and flow control performed by DSS, neither service completely saturates the client link; hence, we also show results for *ttcp*—a simple bandwidth testing tool that sends fabricated data and is able to attain 97% of the theoretical maximum throughput (1128Kb/s) on a 10Mbps link with our specific TCP/IP configuration. Samba and ttcp results are further divided into *incoming* and *outgoing* transfers (from server's point of view) because aggregate throughputs in these two situations differ considerably. We use "in" and "out" to denote the transfer direction from now on.

The first thing to note is that the throughput of services under FT-TCP is either statistically indistinguishable from or only slightly lower than the throughput under clean TCP. The worst relative overhead is about 1.8% for *Samba in*. The overheads for cold and hot backups are statistically indistinguishable. We were expecting faster throughput with a cold backup since a hot one does all the work that a cold one does (i.e. buffers requests) and more, but apparently the additional CPU load on the backup was not sufficient to slow down the buffering process.

DSS connection is the least affected by FT-TCP because it throttles itself down to a low throughput of about 20Kb/s (appropriate for streaming media over a modem connection), leaving plenty of time between `send()` calls to absorb the extra latency of FT-TCP. On the other hand, Samba is affected the most because it performs a larger number of syscalls in general (e.g. for a 4Mb incoming transfer Samba executes approximately 4,870 syscalls, while ttcp executes approximately 2,940).

Finally, there is a marked difference between *in* and *out* throughput values for both ttcp and Samba. Much of the difference is because one rarely gets identical performance from TCP in both directions of the same physical link when the endpoints don't have the same configuration. Differences in hardware and operating systems affect the dynamics of connections and lead to significant (in our case around 10% for ttcp and 17% for Samba) differences in performance. The additional overhead in Samba out is probably due to disk caching—with a cold cache every file read hits the disk, but a series of writes can be absorbed by the cache.

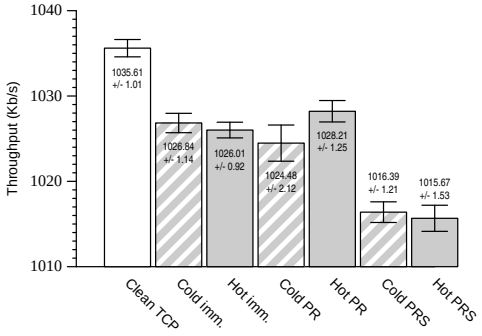

Figure 3. *Incoming Samba throughput*

To evaluate the overhead of interception and buffering more precisely, we plotted all measured *Samba in* throughput values in Figure 3 (note that the Y axis does not start at 0 so as to make the comparison of values easier). Results from other experiments showed a similar pattern, but the overhead differences of different modes were more apparent in this data set. There are two things worth attention on this graph. First, with the exception of *PR*, throughputs with cold and hot backups are statistically indistinguishable. Second, once again *Hot PR* aside, throughput values decrease as we go from *Clean TCP*, to *immediate*, to *PR*, and to *PRS*. This was expected since with each step additional work and buffering are performed by machines running FT-TCP. The throughput of *Hot PR* was higher than we expected. We don't have a good explanation for this, so the matter requires further investigation. In any case, the differences among all the modes are relatively small and our main conclusion about FT-TCP overhead being small is unaffected by this anomaly.

Concurrent client connections compete for access to internal FT-TCP data structures and to the private communication channel between replicas. To see whether this contention was a significant source of overhead, we measured per-client throughput while increasing the number of concurrent connections. We configured ttcp clients to perform an incoming transfer at the rate of 50Kb/sec so we could run at least 20 clients without saturating the 1Mbps link. With both *Clean TCP* and with FT-TCP all clients were able to maintain 50Kb/sec throughput until the number of clients exceeded 20, at which point the link became the bottleneck and the throughput seen by each client dropped.

5.3 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 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 87-byte request and the 49-byte reply. We also measured *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 1 with minimum, mean, and maximum values, as well as their standard deviations. There were 30 Samba request latency measurements, 68 packet latency measurements, and 230 buffering latency measurements (which include both packet and syscall requests).

The average Samba request latency almost tripled (from 2.2 ms to 5.8 ms under cold PRS and 6.2 ms under hot PRS) when FT-TCP was added. Although that may seem like a significant increase in latency, the values are still low enough that from the human perspective responsiveness is not affected at all. Some of this can be attributed to the increase in packet latency that is shown in the next column. Keep in mind that our Samba request consists of two incoming and two outgoing data packets along with some ACKs, so it's not directly comparable with TCP packet latency. It also approximately tripled from 0.7 ms to around 2.3 ms due to interception and buffering overhead. While such an increase may be significant in some circumstances, such latencies are comparable to connection latencies experienced across a WAN. For transfers that saturated the link and used mostly full-sized packets (1,460 data bytes)—such as *ttcp in* and *Samba in*—the latency of packets for both Clean TCP and FT-TCP connections was around 6 ms, which is consistent with the values we reported previously [2].

The values of FT-TCP buffering latencies in the third column are interesting for a couple of reasons. For one thing, they offer us another way to quantify the difference between a cold and a hot backup. As far as primary is concerned, the only difference is the extra 30 microsecond buffering delay on average. This is the reason cold throughput results in the previous section are slightly higher than hot results. The minimal buffering latencies are also useful for placing a lower bound on the round-trip times for messages between our replicas. The RTT is useful for determining reasonable values for the failure detection mechanism described in the next section.

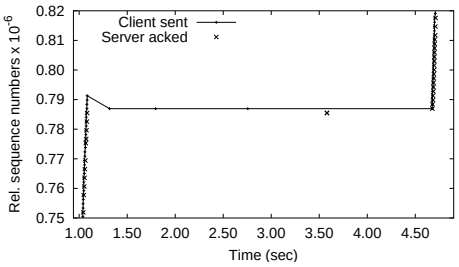

Figure 4. *Behavior of FT-TCP for a long (2.5 sec) promotion latency with no snooping*

5.4 Failure and Recovery

In our earlier feasibility work [2], we showed that recovery was possible. For this work, we decided to concentrate on understanding 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 fault (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). We've already reported [2] 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 4 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 crashes 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 a very long time—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

Setup	Samba request lat. (μsec)				TCP packet lat. (μsec)				Buffering lat. (μsec)			
	min.	*avg.*	*max.*	σ	*min.*	*avg.*	*max.*	σ	*min.*	*avg.*	*max.*	σ
Clean TCP	2112	2181	2974	154	237	748	3475	593				
Cold PRS	5394	5823	6989	299	827	2162	7759	1186	46	522	1617	244
Hot PRS	5801	6183	7327	260	745	2337	8417	1310	37	551	3689	416

Table 1. *Breakdown of latencies for short Samba requests*

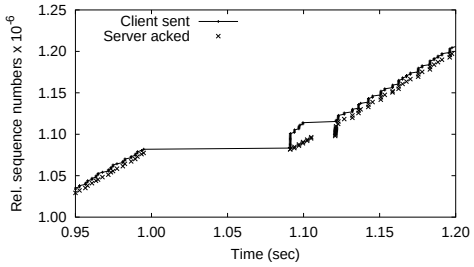

Figure 5. *Behavior of FT-TCP for a short (100 ms) promotion latency with permanent snooping*

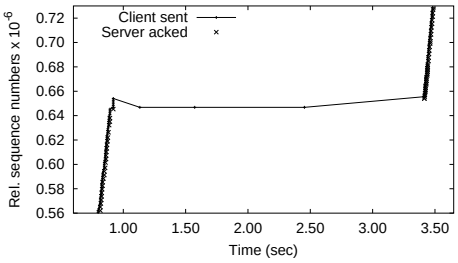

Figure 6. *Behavior of FT-TCP for a long (2.5 sec) promotion latency with permanent or reactive snooping*

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 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 5. With this method the failover time is limited only by the failure detection delay. From Table 1 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 fastest possible failover time, looking at every packet on a busy network may place too heavy of 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 (in fact, there is no harm in starting to snoop whenever a failure is only *suspected* and not necessarily confirmed). 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 restart the data flow as soon as the promotion is complete, as, for example, happens around 3.4 seconds in Figure 6. 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 [9]. In [5] it was used for primary-backup replication of a network file system service. A fault-tolerant TCP system described in [8] also relies on permanent snooping to obtain client packets on the backup.

6 Conclusion

Our earlier work on FT-TCP [2] demonstrated the feasibility of a server-side recovery approach to masking the fail-

ure of a TCP-based server from its clients. In this paper, we addressed the question of how it performed in practice. To do so, we applied FT-TCP to two existing services—Samba and DSS—and determined its impact for both failure-free execution and for executions with failures. We also examined how to reduce the failover time when recovering a TCP connection. We chose as an embodiment a configuration in which the network link between the primary server and its backups was fast, since this is by far the most common configuration used in practice. We found that:

- 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 failure-free overhead of FT-TCP is very low and the system does not impose any new scalability problems. The maximum throughput overhead that we found was for large file transfers to a Samba server. Such requests put a heavy load on the buffering mechanism by sending it a large number of syscalls and packets. Even so, the overhead was under 2%.
- The performance of a hot backup with FT-TCP is nearly indistinguishable from the performance of a cold backup. For cold backups, we did not checkpoint the service (meaning that it would be recovered from its initial state); checkpointing could have a large impact on server performance.
- 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.

We have only looked at two services, and they are similar in that they do not impose a large computational overhead on the server processor. We are interested in the case where the server does have a large computational overhead, but such services are less common in practice.

References

[1] N. Aghdaie and Y. Tamir. Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In *Proc. IEEE Intl. Conf. on Computer Communications and Networks*, 2002.

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

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

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

[5] 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, 1994.

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

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

[8] M. Orgiyan and C. Fetzer. Tapping TCP streams. In *Proc. IEEE Intl. Symp. on Network Computing and Applications (NCA2001)*, 2002.

[9] M. Powell and D. Presotto. Publishing: a reliable broadcast communication mechanism. In *Proc. Symp. on Operating Systems Principles*, pages 100–109, 1983.

[10] G. Shenoy, S. Satapati, and R. Bettati. HydraNet-FT: Network support for dependable services. In *Proc. 20th Intl. Conf. on Distributed Computing Systems*, 2000.

[11] A. Snoeren, D. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. 3rd USENIX Symp. on Internet Technologies and Systems (USITS)*, pages 97–108, 2001.

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

[13] V. Zandy and B. Miller. Reliable network connections. In *Proc. ACM MobiCom*, 2002.