Wrapping Server-Side TCP to Mask Connection Failures

Lorenzo Alvisi, Thomas C. Bressoud, Ayman El-Khashab, Keith Marzullo, Dmitrii Zagorodnov

Abstract— **We present an implementation of a fault-tolerant TCP (FT-TCP) that allows a faulty server to keep its TCP connections open until it either recovers or it is failed over to a backup. The failure and recovery of the server process are completely transparent to client processes connected with it via TCP. FT-TCP does not affect the software running on a client, does not require to change the server's TCP implementation, and does not use a proxy.**

Keywords—**Fault-tolerance, TCP, Rollback-Recovery.**

I. INTRODUCTION

WHEN processes on different processors communicate, they most often do so using TCP. TCP, which provides a bi-directional byte stream, is used for short lived sessions like those commonly used with HTTP, for long lived sessions like large file transfers, and for continuous sessions like those used with BGP [8]. TCP is exceptionally well engineered; literally man-millennia have gone into the design, implementation, performance tuning and enhancement of TCP.

Consider a TCP session set up between two processes, one of which is a *client* and the other a *server*. For our purposes, the difference between the two is a question of deployment and control: an organization is responsible for the server, but clients are associated with individuals that may not be part of that organization.[1] This system is distributed, and can suffer from the failure of the client or the server. The failure of the client is out of the control of the organization, but the failure of the server is not. If the server provides some service from which the organization earns money, then recovery of the server is important. Recovering the state of the server's application can be done by checkpointing the application's state and restarting from this checkpoint on either the same or a different processor. But, the TCP session will need to be recovered as well. We assume that rewriting the client and server applications to detect TCP session loss and to reestablish the session as necessary is not a feasible approach because of the cost and danger of introducing bugs into the application.

One approach to recovering the TCP session is to insert a layer of software between the TCP layer and the application layer of the client, and to insert a similar layer between the TCP layer and the application layer of the server. Such a layer provides the TCP abstractions as well as recovers a lost TCP session. To do the latter, this layer implements some kind of checkpointing of the TCP connection state. It also re-establishes the connection between the old client and the new server in a state consistent with the state of the connection when the old server crashed. This is not a trivial layer of software to develop, especially when the failure-free performance of the connection is an issue [7]. The primary drawback of this approach, though, is that the required layer of software must be run by both the server and the clients. For some applications this is not a problem, but in general clients are out of the control of the organization that maintains the servers.

A second approach is to redesign the TCP layer on the server to add support for checkpointing and restarting to the TCP implementation. This redesigned TCP layer checkpoints the state of the connection so that the new server sends packets consistent with the state of the connection when the old server crashed. For example, the TCP protocol requires each end of a connection to choose fresh initial sequence numbers when opening a connection. The redesigned TCP layer would need to re-open the connection using the old initial sequence numbers. This redesigned TCP layer is also not trivial to implement, especially when failure-free performance and security of the connection are important. The primary drawback with this approach, though, is that it requires a new TCP implementation on the server side. At best, the support would be retrofitted into an existing implementation, and so would

Lorenzo Alvisi is with the Department of Computer Sciences, UT Austin. He is supported in part by the National Science Foundation (CAREER award CCR-9734185, Research Infrastructure Award CDA-9734185) and DARPA/SPAWAR grant N66001-98-8911.

Thomas C. Bressoud is with Lucent Technologies Bell Laboratories.

Ayman El-Khashab is with the Department of Electrical and Computer Engineering, UT Austin.

Keith Marzullo is with the Department of Computer Science and Engineering University of California, San Diego. He is supported in part by the National Science Foundation NSF CCR-9803743 and DARPA/SPAWAR grant N66001-98-8911.

Dmitrii Zagorodnov is with the Department of Computer Science and Engineering University of California, San Diego. He is supported in part by DARPA/SPAWAR grant N66001-98-8911.

[1] With our terminology a server can do both an active and a passive open.

need to be re-retrofitted whenever the TCP implementation is improved or enhanced. Such repeated retrofitting exacts a high development and maintenance cost.

A third approach is to leave the client and server code untouched, and to redirect all TCP traffic between them through a *proxy*. This approach generalizes the work described in [5] that uses such a proxy for mobile computing. The proxy maintains the state of the connection between the client and the server. If the server crashes, then the proxy switches the connection to an alternate server and ensures that the new connection is consistent with the client's state. Like the second approach, this approach needs to ensure the sequence numbers of the new connection are consistent with those of the old connection. The main drawback of this approach is that it introduces a new single point of failure, namely the proxy. This single point of failure is especially troublesome when a proxy is shared among many servers. And, when the proxy fails, the TCP connection between the client and the proxy needs to be made fault-tolerant, which raises the original problem again.

The approach that we develop in this paper does not suffer from the drawbacks of the previous approaches. We present an implementation of a fault-tolerant TCP that does not affect the software running on a client, does not cause the server's TCP implementation to be changed, and does not use a proxy. With Fault-Tolerant TCP (FT-TCP) a faulty process can keep its TCP connections open until it either recovers or it is failed over to a backup. No client process connected with a crashed process running FT-TCP can detect any anomaly in the behavior of their TCP connections: the failure and recovery of the crashed process are completely transparent. We show that for some reasonable network configurations, our approach has a negligible impact on both throughput and latency.

Although the details of the solution that we outline are specific to TCP, the architecture that we propose is sufficiently general to be applicable in principle to other connection-oriented network protocols.

The remainder of this paper is organized as follows. In Section II we give an overview of the major architectural components of FT-TCP. This is followed by a discussion of the recovery of logged data in Section III. In Section IV we describe the FT-TCP protocol, and its operation during failure-free executions and recovery. Section V presents an empirical evaluation of the performance of FT-TCP. Section VI discusses further implementation issues, and Section VII concludes the paper.

Because of lack of space, we omit reviewing TCP. We assume that the reader is familiar with how connections are opened and closed, how TCP represents the sliding window, and how flow control is implemented.

II. Architecture

FT-TCP is based on the concept of *wrapping*, in which a layer of software surrounds the TCP layer and intercepts all communication with that layer. The communication can come from either the IP layer upon which the TCP layer is built (the corresponding wrapper is called *the south side wrap*, or *SSW* for short) and from the application that uses the TCP layer to read and write data (the corresponding layer is called *the north side wrap*, or *NSW* for short). These two wraps in turn communicate with a *logger*. The resulting architecture is illustrated in Figure 1. Together, these three components maintain at the logger the current state of the TCP connection. And, if the current TCP connection goes down, they cooperate to restart the server, to restore the state of the TCP connection to what it had before the crash, and to map the TCP sequence numbers used with the client state to the TCP sequence numbers used with the current server state and vice versa.

Given a TCP connection between a client and a server, 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*.

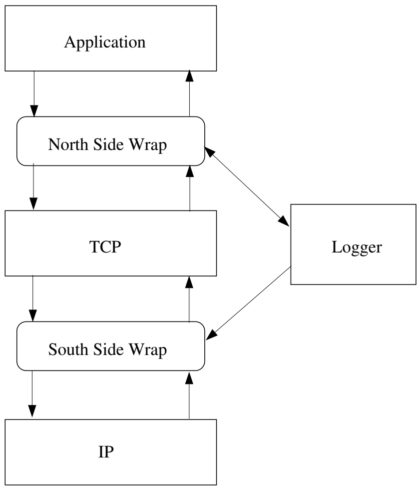

Fig. 1. FT-TCP architecture.

SSW intercepts data passing between the TCP layer and the IP layer. For segments coming from the TCP layer to the IP layer, SSW maps the sequence number from the client's connection state to the server's current connection state. It does so to allow a recovering server's TCP to propose a new initial sequence number during the three-way handshake at connection establishment; SSW translates the sequence numbers to be consistent with those used in the original handshake. For packets going from the IP layer to the TCP layer, SSW performs the inverse mapping on the ACK number. SSW also sends packets to the logger and either modifies or generates acks coming from the server to the client. It does so to ensure that the client side discards data from its send buffer only after it has been logged at the logger.

NSW intercepts read and write socket calls from the application to the TCP layer. During normal operation, NSW logs the amount of data that is returned with each read socket call. We call this value the *read length* for that socket call. When a crashed server is recovered, NSW forces read socket calls to have the same data and read lengths. It does so to ensure deterministic recovery, as we discuss in Section III. During recovery NSW also discards write socket calls to avoid resending data to the client.

The logger runs on a processor that fails independently from the server. It stores information for recovery purposes. In particular, it logs the connection state information (such as the advertized window size and the acknowledgment sequence number), the data, and the read lengths. The logger acknowledges to the north and south side wraps after logging data.[2]

Checkpointing the state of the server running on the server is outside of the scope of this paper. Hence, in this paper we assume that a restarting server has the application restart from its initial state. We assume that the process issues the same sequence of read socket calls when replayed as long as the read lengths of each read socket call has the same value as before. We also assume that there is only a single TCP stream open with the server. The generalization of the ideas given in this paper to a server supporting multiple streams is not hard, but having multiple streams often implies a multithreaded server. With a multithreaded server, the interdigitation of stream reads by different threads needs to be addressed. Again, this issue is outside of the scope of this paper.

Our protocol requires a mechanism that allows a process on another processor to take over the IP address of a process on a failed processor. This mechanism also updates the ARP cache of any client on the same physical network as the failed or recovered server. The latter can be done using gratuitous ARP. See [9] for further details and related issues.

III. Recovering from Logged Data

FT-TCP recovers the state of a crashed server from the recovery data on the logger. Hence, it is necessary that the logger store the latest state of the server, namely all of the packets it has received and the read lengths it has generated. Waiting for the recovery data to be stored at the logger, however, incurs a prohibitively large latency, and so FT-TCP sends recovery data to the logger asynchronously. Because some recovery data may be lost when a crash occurs, this asynchronous logging of data can restore the server to a state that is earlier than the state that the client knows the server had attained before crashing.

For example, consider the connection state information. The ACK sequence number of a packet from the server indicates the amount of data that the server has received from the client. Suppose that the server sends a packet with an ACK sequence number *asn* but the logger has stored data only through $asn - \ell$ for $\ell > 1$. When the server recovers, the TCP layer knows of data only through $asn - \ell$, and the next packet it sends has an ACK sequence number less than *asn*. To recover the server farther, the client's TCP layer would need to send the missing data. Since the server has already acknowledged through *asn*, however, the client may have discarded this data from its send buffers. To avoid this problem, SSW never allows the ACK sequence number of an outgoing segment to be larger than $asn - \ell + 1$.

A similar problem occurs with respect to the data exchanged between the client and server applications. The state of the server application may depend on the read lengths the application observes. For example, suppose the application attempts to read 8,000 bytes. It may take a different action if the `read` socket call returns less than 1,000 bytes as compared to it returning at least 1,000 bytes. It is because of such a possibility that NSW records read lengths.

Suppose now that a `read` returns 900 bytes. NSW sends this read length to the logger, but the server crashes before the logger receives this message. Thus, the read length is lost. After restarting, the `read` is re-executed, but, because more data has become available in the receive buffers of TCP, the new `read` returns 1,500 bytes, bringing the server to a state inconsistent with what it had before the crash. If the client can observe this inconsistency—for example, by receiving data from the server both before it crashed (reflecting the state in which the server read only 900 bytes) and after it crashed (reflecting the state in which the server read 1,500 bytes) then the failure of the server is not masked. If the only way in which the client can observe the state of the server is by receiving data from it, then this problem can be solved by delaying all `write` socket calls by the server application until all prior read lengths are known to be stored on the logger. This is what FT-TCP does.

The problem of restoring the crashed server to a state consistent with its last state observed by a client is an in-

[2] We use the term *ack* to refer to a TCP segment that acknowledges the receipt of data, and use the term *acknowledgement* to refer to a message that the logger sends to the server indicating that data is logged.

stance of the more general *Output Commit* problem [2]. We discuss the Output Commit problem further in Section VI.

IV. PROTOCOL

We now describe the operation of FT-TCP. After introducing the state that FT-TCP maintains, we describe how an FT-TCP connection is set up. We then describe the operation of SSW and NSW while the TCP connection is open and operational, either for the first time or after a failed server has recovered. We call this mode *normal operation*. We then describe how a failed server recovers its TCP stream state.

A. Variables

FT-TCP maintains the following variables:

- `delta_seq` This variable allows SSW to map sequence numbers for the outstream between the server's TCP layer and the client's TCP layer.
- `stable_seq` This variable is the smallest sequence number of the instream that SSW does not know is stored on the logger. Note that this value can never be larger (ignoring 32-bit wrap) than the largest sequence number that the server has received from the client. This variable is also computed during recovery from the data stored on the logger.
- `server_seq` This variable is the highest sequence number of the outstream that SSW knows to have been acknowledged by the client. This variable can be computed during recovery from the data stored on the logger.
- `unstable_reads` This variable counts the number of read socket calls whose read lengths NSW does not know to be recorded by the logger. If `unstable_reads` is zero, then NSW knows that the logger has recorded the read lengths of all prior read socket calls.
- `restarting` This boolean is `true` while the server is not in normal operation.

B. Opening the Initial Connection

When the connection is initially established, the only action FT-TCP takes is to capture and log both the client's and the server's initial sequence numbers. SSW does not pass the segment that acknowledges the client's SYN to the server's IP layer until the logger acknowledges that these initial sequence numbers are logged. If it did not do this, the client might believe a connection is established that a failure and recovery might not be aware of.

SSW completes the initialization of FT-TCP by setting `delta_seq` to zero, `stable_seq` to the client's initial sequence number plus one, `unstable_reads` to zero, and `restarting` to `false`.

C. Normal Operation of SSW

During normal operation, SSW responds to three different events: receiving a packet from IP, receiving a segment from the TCP layer, and the receipt of an acknowledgement from the logger.

When SSW receives a packet from IP, it first forwards the packet to the logger. SSW then subtracts `delta_seq` from the ACK number. Since doing so changes the payload, 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 ACK number, and the new ACK number. SSW then passes the result to the TCP layer, without waiting for an acknowledgement from the logger indicating that the packet has been logged.

When SSW receives an acknowledgement from the logger for a packet, SSW updates `stable_seq` if necessary. Specifically, if the acknowledgement 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 its current value and $sn + \ell + 1$.

When SSW receives a segment from the TCP layer, it remaps the sequence number by adding `delta_seq` to it. SSW then sets the ACK number to `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 of *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, 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 IP.

Figure 2 illustrates normal operation for SSW as described above.

D. Normal Operation of NSW

During normal operation, NSW reponds to three different events: a `read` socket call, a `write` socket call, and receiving an acknowledgement from the logger.

For each `read`, NSW sends the resulting read length to logger and increments `unstable_reads`. When NSW receives the acknowledgement from the logger, it decrements `unstable_reads`. And, for each `write`, NSW blocks the call until `unstable_reads` is zero.

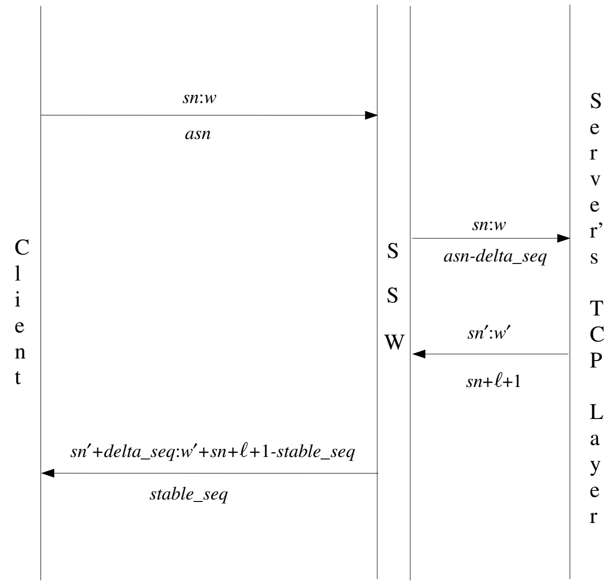

Fig. 2. Normal operation of SSW.

E. Re-establishing a Connection

The following steps occur when the server crashes:

1. The logger detects the failure of the server. It temporarily takes over the role of the server by responding to instream packets with a TCP segment. This segment has a closed window and acks the data received from the client up to the last value the logger has for `stable_seq`.
2. The server restarts and FT-TCP reconnects with the log server. FT-TCP sets `stable_seq` and `server_seq` to the values computed by the logger based on the logged data. `unstable_reads` is set to zero and `recovering` is set to `true`.
Once the server obtains and acknowledges this information from the log server, the logger implicitly relinquishes the generation of closed window acknowledgements to SSW. SSW continues to generate periodically these acknowledgements as long as `recovering` is `true`.
3. The restarting application executes either an `accept` or `connect` socket call. We describe here only what happens if `accept` is called; `connect` is handled similarly. NSW has SSW fabricate a SYN that appears to come from the client. This SYN has an initial sequence number of `stable_seq`. Thus, the server's TCP layer will start accepting client data with sequence number `stable_seq` plus one. SSW passes this SYN to the TCP layer.
4. The acknowledging SYN generated by the server's TCP layer is captured by SSW. SSW sets `delta_seq` to the logged initial server sequence number minus the server's TCP layer new proposed initial sequence number. SSW discards this segment, fabricates the corresponding ack, and passes the ack to the server's TCP layer.
5. The server's application starts running as part of the restart of the server. Every `read` socket call it executes is captured by NSW, which supplies the corresponding data from the logger. The amount of data returned with each `read` is determined by the corresponding logged read length.
Every `write` socket call is also captured by NSW. NSW keeps a running total of the number of bytes that have been written by the server since starting recovery. As long as each write produces data that was written before (as determined from the logged initial client sequence number and `server_seq`), NSW discards the write and returns a successful `write` completion to the application.
Once the last logged `read` is replayed and the server's application has written all of the bytes it had written before, NSW sets `recovering` to `false`, and the server resumes the normal mode of operation. This may cause a `read` socket call to block until all the replayed writes have occurred. It may also cause data to be rewritten to the outstream after `recovering` is `true`. In the latter case, the resulting TCP segments will be discarded by the client's TCP layer as being delayed duplicates.

F. Ack Strategies

As described Section IV-C, SSW modifies acks in the outstream to ensure that the client does not discard instream data before the SSW knows it is logged on the logger. It does so by ensuring that the ack sequence number is never larger than `stable_seq`. All outstream segments are immediately processed and passed to IP. Further, no additional segments are generated by SSW. We call this the *Basic* ack strategy. By itself, Basic is not a satisfactory strategy.

To make this concrete, assume that a segment S arrives at SSW in the instream carrying bytes starting with sequence number sn. SSW sends this data to the logger, but by the time the server's TCP layer generates an ack for it, the logger has not yet acknowledged it, meaning that `stable_seq` is still less than sn. Even if the acknowledgement from the logger 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). Worse occurs, however, when the outstream traffic is low and the instream traffic is blocked due to windowing restrictions. For example, consider what happens when slow start [4] 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 on the logger, 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.

Two simple ack strategies that avoid such problems are:
Lazy: SSW uses the Basic ack strategy for outstream segments that carry data. Segments that carry no data (and hence are only acking the delivery of instream data) are instead held by SSW until their ack sequence number is at least `stable_seq`.
Eager: SSW uses the Basic ack strategy. In addition, SSW generates an ack on the outstream for every acknowledgement it receives from the logger, thus acking every instream packet.

The Eager strategy can significantly increase bandwidth demand. A fourth ack strategy, which we call *Conditional*, addresses this drawback. It is a variant of Eager for which only some acknowledgements from the logger cause SSW to generate an ack. Specifically, consider the point SSW receives an acknowledgement from the logger for a packet S. SSW generates an ack iff, given that TCP has attempted an ack for S, and during the interval from the arrival of S at SSW to the arrival of the acknowledgement of S, SSW receives no packets from the IP layer.

The performance of FT-TCP using only Basic is quite poor; we have found that the resulting FT-TCP is often unable to sustain a bulk-transfer connection. We discuss the performance of the other three ack strategies in Section V.

G. Logger

The server and the logger communicate using TCP. The server sends a segment to the logger for every segment it receives from the client. Each such segment generates an acknowledgement from the logger, which is simply a 32-bit integer. This difference in the amount of traffic raises the question of whether Nagle's algorithm [6] should be enabled for the stream from the logger to the server. Enabling Nagle results in batching multiple acknowledgements into a single segment and therefore reduces the load on the network connecting the server and the logger. If this network is the same one that the client is on, then the extra packet overhead incurred using TCP with Nagle disabled may significantly reduce the bandwidth of the FT-TCP connection. We explore this question in Section V.

Each acknowledgement from the logger is simply a sequence number: it is the lowest sequence number of client data that is not logged. Thus, the sequence of acknowledgements is monotonically increasing (ignoring the 32 bit wrap). This means that the last acknowledgement in any batch contained in a segment is the only one that needs to be processed by SSW, since it dominates the other acknowledgements. We have found, though, that the overhead incurred by having SSW process each acknowledgement is small enough that it is not worth taking advantage of this observation. Note that with the conditional ack strategy, the only acknowledgement in such a batch that could cause SSW to generate an ack (as discussed in Section IV-F) is the last one. Hence, the effect of processing the other acknowledgements is just to increase `stable_seq`.

V. Performance

In this section, we first describe the metrics of interest and the experimental setup. We then present the results of our experiments.

A. Goals and Experiments

To show that FT-TCP is viable in practice, we evaluated a prototype implementation of FT-TCP. Specifically, we used an application in which the client transmits a stream, as bulk data, to the server, as fast as it can. The server simply discards this data. We measured:
1. The throughput of FT-TCP as compared to the throughput for the same bulk data transfer of an unwrapped TCP layer at the server.
2. The additional latency introduced by FT-TCP.
3. The recovery time of the server, divided into its constituent parts.

We chose bulk transfer from the client to the server for two reasons. First, by simple inspection of the protocol, it is clear that the outstream incurs a much smaller overhead when compared with the instream. Second, bulk data transfer with no server computation is the most disadvantageous workload for FT-TCP; we expect that with other workload types, FT-TCP would compare more favorably with TCP.

We ran our experiments using three separate machines, one each for the client, the server, and the logger. On the server, we implemented Linux kernel modules for NSW and SSW, a Unix application providing communication between the kernel modules and the logger, and the server application. We implemented corresponding applications on the client and the logger.

Our server was a 450 MHz Pentium II workstation with 512 KB cache and 128MB of memory, while both the client and the logger ran on 300 MHz Pentium IIs with 512 KB cache and 64 MB of memory. The server allowed connection to the client and the logger via two separate 100 Mbps Ethernet adaptors (both Intel EtherExpress Pro 100). The client used a 10 Mbps 3Com Megahertz PCMCIA Ethernet card and the logger used a 100 Mbps 3Com PCI Etherlink XL. FT-TCP wrappers were implemented for Linux kernel 2.0.36, while Linux 2.2.x kernels were run on the client and logger machines.

The `tcpdump` utility was used to collect timestamps and packet information for the connections under test. This was executed on the client machine to get accurate client-side measurements of latency of FT-TCP connections.

To measure throughput and latency, we ran our experiments with three different network configurations:

1. The client and server share one 10 MB Ethernet and the server and logger share another 10 MB Ethernet. We call this configuration *10-10*.
2. The client, server, and logger are all on the same 10 MB Ethernet. We call this configuration *10 Shared*.
3. The client and server share one 10 MB Ethernet and the server and the logger share a 100 MB Ethernet. We call this configuration *10-100*.

The first configuration models a simple network setup. The second configuration allows us to determine the impact of having all three system components share the same network. The third configuration allows us to remove the network bandwidth between the server and the logger as the (expected) bottleneck.

To measure recovery time, we timed how long it takes to recover the server from the data on the logger. Since the client does not participate in recovery, for these experiments we considered only the two configurations *10-10* and *10-100*.

B. Results

The results below are for a 1MB transfer from client to server. We first gathered the results of a non-wrapped TCP stack at the server, with the same client and server applications as used for the experimental runs.

For each network configuration and ack strategy, we gathered results from 12 runs. To measure throughput, we applied a simple linear least squares fit with the independent variable being the beginning sequence number of a segment and the dependent variable being the time this segment was received by SSW. The coefficient of determination R^2 for all but one of these fits is 0.99 or better; R^2 for the remaining one (Lazy for 10 Shared) is 0.83. The slope of a least squares fit provides an accurate representation of the throughput of the connection. We present the error bounds of these slopes for a 95% confidence interval.

We measured latency by post-processing the `tcpdump` results to determine the interval from the time data in the instream was sent by the client to the time the ack for that data was received by the client. We averaged these intervals and calculated 95% confidence intervals.

B.1 10-10

Table I presents the results for the 10-10 configuration giving throughput, average latency, and ack count for an unwrapped TCP (*Clean*) and for each ack strategy.

TABLE I
10-10 PERFORMANCE.

	Throughput (KB/s)	Error Bound (KB/s)	% of Clean	Avg. Latency (ms)	Error Bound (ms)	Ack Count
Clean	1007.69	0.53	100.00%	5.82	3.39	3064
Lazy	241.33	0.76	23.95%	27.75	17.23	1766
Lazy64k	488.08	3.26	48.44%	44.89	29.75	555
Eager	722.84	1.16	71.73%	56.70	35.77	12810
Cond.	651.02	2.50	64.61%	53.96	58.46	3276

From the table, we first note that the throughput of the Lazy ack strategy is only 24% of that of unwrapped TCP. The explanation for this results is attained through comparison with unwrapped TCP as observed through the `tcpdump` logs of the runs.

Under unwrapped TCP, 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, $x+1$, $x+2$. At this point in the interleaving, the server sends an ack for the bytes in $x-1$, which allows the client to send packets $x+3$, $x+4$, $x+5$, 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 logger. This pattern of behavior is indicative of a fast sender and a slow receiver. Note the reduced number of acknowledgments over the 12 runs. With the additional latencies, the processing of FT-TCP, and the communication between server and logger, the server application is no longer consuming data as fast as the client is sending it.

When we tried to address the problem by increasing the receive buffer size to 64K bytes (indicated in table entry *Lazy64K*), we found that the pattern of burst of data followed by acks persisted, but its effect was reduced at a cost of increased average latency[3]. The Eager ack strategy generates acks more aggressively and so it helps break

[3]The average latency is indicative of the size of the window and the number of unacknowledged packets in the window. The standard de-

the fast-sender-and-slow-receiver pattern. This comes at a cost of increased bandwidth for the additional acks. The Conditional ack strategy also helps break the pattern, but does better than Eager since it does not generate as many acks.

B.2 10 Shared

In this network configuration, the same 10 Mbps network segment is utilized both by the client to server communication and by the server to logger communication. Because of this, at least twice as many data bits are transmitted across the shared medium. Since the client and unwrapped TCP server are close to saturating the 10 Mbps link, FT-TCP can only be expected to perform no better than 50% of clean TCP. In addition, there is increased contention for the CSMA/CD physical link and the corresponding backoffs.

The results in Table II show that Eager and Conditional provide approximately a third of unwrapped TCP performance. Conditional achieves this by sending 3,500 acks, while Eager sends almost 13,000 acks. These additional acks take up additional bandwidth and degrade performance slightly. In this configuration, Lazy suffers from the same performance-draining burst interleaving that it did with the previous network configuration. And, as with 10-10, when we increased the receive buffer size, the same pattern developed, but with reduced effects.

TABLE II
10 SHARED PERFORMANCE.

	Throughput (KB/s)	Error Bound (KB/s)	% of Clean	Avg. Latency (ms)	Error Bound (ms)	Ack Count
Clean	1007.69	0.53	100.00%	5.82	3.39	3064
Lazy	195.17	3.79	19.37%	29.15	20.75	2100
Lazy64k	321.20	1.39	31.88%	72.06	43.92	559
Eager	338.50	0.60	33.59%	89.35	108.13	12946
Cond.	343.41	0.73	34.08%	89.47	92.73	3515

Under the Lazy ack strategy, the error bound on throughput is noticeably worse. Further, recall from the beginning of Section V-B that the coefficient of determination R^2 on a linear least squares fit for Lazy with 10 Shared was 0.83. The cause for the relatively large residual variance is that the set of 12 runs exhibits a bimodal pattern: some of the runs complete the 1 MB transfer in just over 3 seconds while the others take over 5 seconds. Figure 3 illustrates this bimodality where a representative efficient Lazy run is labeled *Lazy Good* and a representative inefficient run is labeled *Lazy Bad*. For both completeness and comparison, the figure also includes representative runs for unwrapped TCP, Eager, and Conditional. The Lazy Bad run exhibits the pattern of fast sender, slow server as discussed in the previous section.

viation also increases since a single acknowledgment acks data from multiple packets in the window.

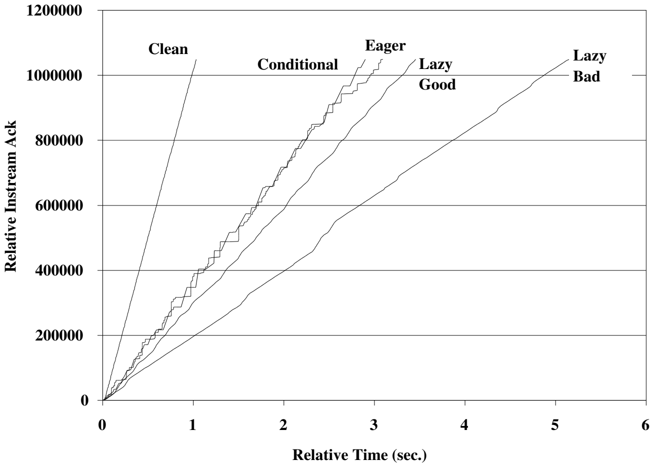

Fig. 3. 10 Shared Acks vs. Time.

A closer look at these runs illustrate the ack pattern and ack frequency of each of the strategies. Figure 4 presents the stream relative acks as a function of time for the first 200 ms of the same sample runs shown in Figure 3. For all of the runs, the first 20 ms are consumed in getting from the point of connection establishment to the first instream data ack. Clean shows a nearly ideal pattern of acks. For Eager, many of acks are redundant and come in bursts, resulting in the staircase pattern of the curve. Conditional moderates this effect with fewer acks, and, for Lazy Bad, the acks are very infrequent.

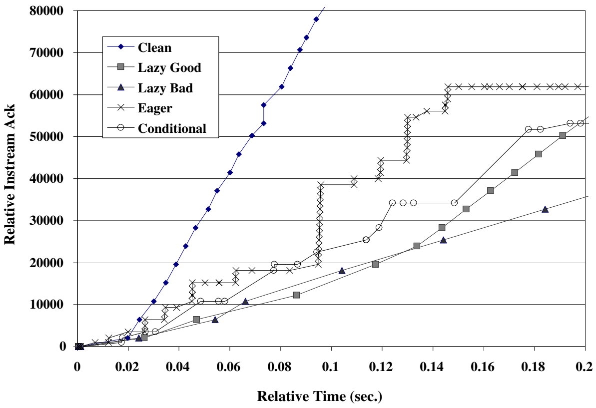

Fig. 4. 10 Shared Acks vs. Time for a Run Prefix.

B.3 10-100

As with 10-10, this network configuration separates the network segments used for client to server communication and server to logger communication. Further, the server to logger communication is over a faster link. The effect of the faster back-end link is to reduce the incremental latency of sending data from the server to logger and receiving the acknowledgment from 2-3 ms in the 10 Mbps case to 0.5-0.7 ms.

As Table III shows, reducing the latency makes a striking difference in performance. This seemingly small difference takes us under a threshold so that the receiver (which includes the server application as well as FT-TCP components supporting communication to the logger) is able to keep up with the client.

TABLE III
10-100 PERFORMANCE.

	Throughput (KB/s)	Error Bound (KB/sec)	% of Clean	Avg. Latency (ms)	Error Bound (ms)	Ack Count
Clean	1007.69	0.53	100.00%	5.82	3.39	3064
Lazy	1007.56	0.57	99.99%	5.86	3.14	3082
Lazy64k	929.92	3.91	92.28%	17.48	22.84	1267
Eager	969.84	0.31	96.24%	5.31	3.23	11687
Cond.	985.84	0.38	97.83%	6.03	3.30	5833

For Eager and Conditional, we see the same behavior as before: Eager spends bandwidth to achieve a level of performance that Conditional exceeds by keeping the client well-acknowledged with less than half the number of acks. The clear winner here, however, is Lazy. With such a small additional latency arising from the logger, Lazy sends acks back to the client with the same interleaving pattern with respect to data packets in the window as clean TCP. In doing so, it achieves a performance that is statistically indistinguishable from clean TCP in both throughput and latency.

B.4 Impact of Nagle Algorithm

The measurements given above were all made with the Nagle algorithm enabled on the connection from the logger to the server. The effect on throughput of having the Nagle algorithm disabled on the connection is complex. Table IV summarizes our measurements on this effect. First, as one would expect, the Nagle algorithm has little effect in the 10-100 configuration; the latency is low enough that small segments are rarely delayed. Hence, the number of segments from the logger to the server changes little when Nagle is disabled.

For the other two network configurations, the effect of disabling Nagle can be significant. With Lazy, the additional acknowledgements from the logger cause acks to be more frequently generated, which breaks down the pattern of a fast sender and slow server. In fact, with the Nagle algorithm disabled, Lazy goes from being the worst to the best ack strategy for the 10-shared network configuration in terms of throughput. The impact of disabling Nagle is

TABLE IV
LOGGER NAGLE DISABLED PERFORMANCE.

		Nagle on	Nagle off	change
10-100	**Clean**	1007.69	1007.69	0.0%
	Lazy	1007.56	1007.42	0.0%
	Lazy64k	929.92	896.20	-3.6%
	Eager	969.84	969.11	-0.1%
	Cond	985.84	987.32	0.2%
10-10	**Clean**	1007.69	1007.69	0.0%
	Lazy	241.33	494.22	104.8%
	Lazy64k	488.08	501.08	2.7%
	Eager	722.84	514.72	-28.8%
	Cond	651.02	506.92	-22.1%
10 shared	**Clean**	1007.69	1007.69	0.0%
	Lazy	195.17	368.53	88.8%
	Lazy64k	321.20	321.52	0.1%
	Eager	338.50	274.23	-19.0%
	Cond	343.41	312.72	-8.9%

lost, however, when the server TCP buffering is increased (Lazy64k); the benefit of increased server buffering seems to be close to that of disabling Nagle. For the other two ack strategies, disabling Nagle has a negative effect. Some of this decrease arises from increased contention on the network between the server and logger. Examination of the TCP dump logs, however, hints that more is going on than just increased contention. This point will require deeper research to fully understand.

B.5 Recovery

Table V summarizes the measured recovery times for 20 runs recovering the server application with the full 1 MB of data logged by FT-TCP. We measured both the time from start of recovery to end of recovery and the time required just to play back the data through NSW. From these measurements, we obtained both the restart time, which includes process restart as well as simulating the SYN sequence to the server TCP, and the replay time for the set of reads encountered by NSW.

TABLE V
RECOVERY MEASUREMENTS.

	Restart Time (ms)	Error Bound (ms)	Replay Time (sec.)	Error Bound (sec.)	Avg. Read Time (ms)	Error Bound (ms)
10-10	17.31	0.02	2.571	0.00201	1.893	0.0015
10-100	22.21	13.31	0.485	0.00036	0.338	0.0003
Local	22.44	14.08	0.056	0.00025	0.039	0.0002

As a reference point, the table also includes recovery time from a logger co-located on the server's physical machine (*Local*), thus showing the performance if the back-end network latency were totally eliminated.

Restart time for all three cases is around 20 ms. For both the 10-100 and Local, there was one outlier that took almost a half second to start the process, explaining the higher mean and error bound.

Local gives a lower bound for replay of the 1 MB in-stream at 56 ms. The current recovery implementation uses a synchronous interface between NSW and the logger on each encountered read call. This serialization of recovery data from the logger to NSW explains the low effective throughput relative to the bandwidth of the 10-10 and 10-100 cases. A simple optimization would employ read-ahead by NSW on recovery to achieve much better results.

VI. Other Issues

In this section we discuss some additional implementation issues.

A. Forced Termination of a Server

For most Unix versions, when a process is abnormally terminated the process `exit` routine closes all open TCP streams. For example, if a `SIGKILL` is sent to the server, then the server implicitly closes the connection before terminating. This may result in either an orderly release—the server's TCP starts the close handshake by sending a FIN—or an abortive release—the server sends a RST. Since we test our implementation by terminating the server with a UNIX signal, this implicit close has proven problematic.

We work around this problem by having NSW capture the `close` socket call as well and by using OS platform specific information to identify the close as arising from abnormal process termination (rather than a valid `close` by the server's application). When the close is raised by process cleanup, SSW discards the FIN or RST segment and initiates recovery.

B. Output Commit

In Section III we gave the reasoning for blocking writes to the TCP stream while prior read lengths are not known to be logged. By doing so, we ensure that the client will never know more about the latest state of the server than the logger knows. This approach relies on the (probably valid) assumption that the only communication between the client and the server is via the TCP stream. And, the assumption that read lengths are significant nondeterministic events is weak; we are aware of only contrived servers for which the sequence of bytes written to the outstream is determined by read lengths.

A related issue has to do with consistency between the recovered server and the environment (such as file servers, databases, and physical actuators). A server that changes the environment needs to be written to handle the situation in which the server's failure makes it uncertain upon recovery whether or not the change occurred [3]. The typical approach is to make all changes to the environment idempotent, and have the recovered server redo the change should it be unsure whether the change was indeed made before failure. However, if the server changes the environment before the prior read lengths are stored on the logger, then the server may be unable to recover to the state in which the change occurred. Avoiding this possibility is called the *Output Commit* problem [2].

In FT-TCP, the Output Commit problem can be addressed by preventing all changes to the environment while there are prior read length not known to be logged. Conceptually, this is simple to do; NSW intercepts all such changes and blocks them in the same way it blocks writes to the TCP stream. In practice, this requires to identify a set of system calls that, like `write`, can modify the environment. These calls would be intercepted by NSW.

C. Other Architectures

The architecture we describe for FT-TCP—a logger running on a processor separate from the server—is not the only one worth considering. For example, one could have the logging of packets done on the same processor as the server. The logging could be done asynchronously to disk by using existing techniques for making in-memory logs as reliable as disk [1]. This architecture would decrease the latency of the FT-TCP connection, but would probably not increase the bandwidth. It would also make the only processor that the server could restart on be the original processor.

Another possible architecture implements the logger as a hot standby for the server. The packets sent by SSW would be injected into the SSW of the standby, and the reads of the NSW of the standby would block until the read length from the server arrived. Such an architecture would allow for a small failover time since recovery would just have the TCP stream migrate to the hot standby. The open protocol described in Section IV-E is essentially the protocol that the hot standby runs to set up its TCP stream, and the normal operation protocol for the server is unchanged.

VII. Conclusion

We have described the architecture and performance of FT-TCP, a software that wraps an existing TCP layer to mask server failures from unmodified clients. We have implemented a prototype of FT-TCP and find that, even for a demanding application, it imposes a low overhead on throughput and latency when the connection between the server and the logger is fast.

Further experimentation is called for. For example, we have measured the performance of FT-TCP only for applications with one-way bulk transfer from the client to the server. A study with a wider set of realistic applications would give a better idea of the actual overhead of FT-TCP. And, even with this simple kind of application, we have found that the properties of the communication between the logger and the server have a large and complex influence on the overhead. Finally, we still need to better understand the impact of FT-TCP on the client TCP layer. For example, we have not yet measured the magnitude of the impact of a server crash and recover on the client's RTT measurements.

ACKNOWLEDGMENTS

We would like to thank Susie Armstrong, Jeff Napper, and Phoebe Weidmann for their comments both on the content and on the presentation of this paper.

REFERENCES

[1] P. M. Chen et. al. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. 74–83.

[2] E. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. CMU Technical Report CMU-CS-99-148, June 1999.

[3] J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, Chapter 3, Springer-Verlag Lecture Notes in Computer Science 60, 1978, pp. 393–481.

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

[5] D. Maltz and P. Bhagwat. TCP splicing for application layer proxy performance. IBM Research Report 21139 (Computer Science/Mathematics), IBM Research Division, 17 March 1998.

[6] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.

[7] R. Nasika and P. Dasgupta. Transparent migration of distributed computing processes. In *Proceedings of the Thirteenth International Conference on Parallel and Distributed Computing Systems*, to appear.

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

[9] W. Richard Stevens. TCP/IP illustrated, Volume 1: The protocols. Addison-Wesley 1994.