Handling Catastrophic Failures in Scalable Internet Applications

Michael Haungs
Computer Science Department
California Polytechnic, San Luis Obispo
mhaungs@csc.calpoly.edu

Raju Pandey, Earl Barr
Department of Computer Science
University of California, Davis
email: {pandey, barr}@cs.ucdavis.edu

Abstract

User perceived quality is the most important aspect of Internet applications. After a single negative experience, users tend to switch to one of the other myriad of alternatives available to them on the Internet. Two key components of Internet application quality are scalability and reliability. In this paper, we present the first general-purpose mechanism capable of maintaining reliability in the face of process, machine, and catastrophic failures. We define catastrophic failures as events that cause entire clusters of servers to become unavailable such as network partitioning, router failures, natural disasters, or even terrorist attacks. Our mechanism utilizes client-side tunneling, client-side redirection, and implicit redirection triggers to deliver reliable communication channels. We capitalize on previous work, Redirectable Sockets (RedSocks), that focuses on Internet application scalability. RedSocks are communication channels enhanced with a novel session layer aimed at modernizing network communication. We modify RedSocks to create the first fault tolerant socket solution that can handle all server-side failures. Our mechanism is compatible with NATs and Firewalls, scalable, application independent, and backwards compatible.

1 Introduction

It is crucial for Internet applications to be available 24-7. One study found that two-thirds of Internet users will rarely return to a site after a single bad experience [9]. Bhatti et al. [2] state, "Users have too many web sites that they can use as alternatives if they are either refused entry to one site or are given particulary slow service." For some popular web services, a single server failure can result in tens of thousands of lost customers.

To achieve 24/7 availability, Internet applications need to be scalable and reliable. The most popular solution to scalability is to construct multiple clusters of commodity servers and route incoming requests to them via a mechanism such as DNS Round Robin [3] or URL-rewriting [7]. L4 or L7 switches are then used for intracluster load balancing. No standard solution exists for making Internet applications reliable, but researchers have proposed a number of ideas [1, 11, 13, 8]. One reason for the lack of a standard solution for reliability is the tension that exists between reliability and scalability. Scalable systems are more complex or introduce central failure points which increases the vulnerability of the server system. While overly cautious failover mechanisms directly affect system scalability by increasing server response latency.

Current approaches to Internet application reliability fall into two categories. The first category, *connection redirection*, insures new connections reach a healthy server. Both DNS and network switches, such as Cisco's LocalDirector [4], can be configured to incorporate server health information in their connection routing. Connection redirection complements the second category, *communication fault tolerance*. The goal here is to allow existing communication channels to persist in the face of machine failure. Most solutions use primary-backup or log-based recovery.

However, no solution addresses *catastrophic failures*. We define catastrophic failures as failures that completely block all communication from the current server cluster to the client. These failures can result from accidently cut transmission lines, router hardware failures, natural disasters, or even terrorist attacks. After a catastrophic failure, only the client and a route to an alternate server cluster exists. From this, the existing communication channel must be reconnected to an alternate cluster and all communication state and associated application state reconstructed.

We define a *fault tolerant socket* as a communication channel between two applications, typically a client and a server, that persists in the face of a process, machine, or catastrophic failure. In Figure 1, we illustrate where failures can occur along a communication channel and put them into two categories: *server-side* and *client-side*. Attempts in increasing server application reliability often target process (A) and machine (B) failures. Using a redundant router handles an error at the entry point (D) to the Internet application. Cut cables or Earthquakes are examples of events that can cause the complete failure of server clusters (C) and associated entry points (D). There are two types of failures that occur in the Internet outside the server's administrative domain. The first are errors that can be circumvented via normal IP routing recovery (E). The second (F) occurs when there is no alternative path from the client to the origin server cluster. We categorize failures at points A through F as server-side errors. Fault tolerant sockets must handle all server-side failures. Client-side failures (H, I, and J in Figure 1) are beyond the scope of Internet server applications. Put another way, fault tolerant sockets persist as long as the client does.

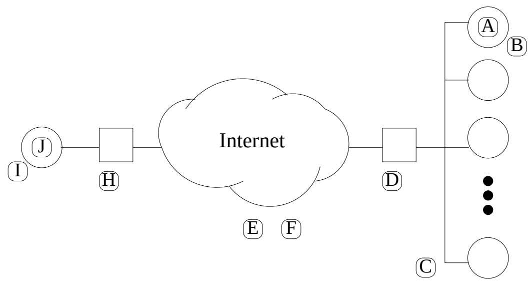


Figure 1. Various failure points along a client/server communication channel. Server-side failures are: (A) Process, (B) Machine, (C) Cluster, (D) Switch, (E) Router with alternate route, and (F) Router with no alternate route. Client-side failures are: (G) Router, (H) Machine, and (I) Process.

To handle server-side errors, a failover mechanism must be able to detect that a failure occured, recover in-flight data, construct a new communication channel to an operational server, and provide a method to synchronize communication and application state. It must do this in a scalable way to be practical. Last, it must be compatible with methods for handling new connection redirection and be application independent.

One way to implement fault tolerant sockets is to use an existing intracluster mechanism and make it an intercluster mechanism. This involves relaying network packets between clusters for every network packet received from a client. This transaction must complete before responding to the client. This exasperates the scalability problem. With this type of solution, you minimally double the amount of traffic seen in individual clusters and increase the server response time by the roundtrip latency between clusters. We do not explore this method.

In this paper, we present our implementation of fault tolerant sockets. We propose a method that relies on client-side support. Client-side support consists of a redirection mechanism (client-side redirection), a method to detect communication channel errors to trigger redirection (implicit redirection triggers), and data synchronization (client-side tunnelling). We first introduced the idea of client-side support in [5] where we discuss using it for connection redirection. We developed RedSocks which uses our session-layer protocol, the Endpoint Operation Protocol (EOP), that allows a server to redirect connections in a flexible, scalable, and application independent way. In this paper, we augment EOP to allow RedSocks to provide communication fault tolerance. Our solution is client transparent, compatible with NAT's and Firewalls, backwards compatible with normal sockets, application independent, handles catastrophic failures, and works with high-performance Internet applications.

In Section 2, we present our fault tolerant socket solution. We follow with a discussion on the details of our implementation. We present experiments, in Section 4, that demonstrate the efficacy of fault tolerant RedSocks. We conclude with a presentation of related work and some parting thoughts.

2 RedSocks Fault Tolerance

We have extended BSD `sockets` with a our session layer protocol, EOP, that generates redirection events and enables endpoint redefinition. We call such sockets *Redirectable sockets* or *RedSocks*. We looked at explicit redirection events in [6] to increase the scalability of internet applications. We now look at RedSocks and implicit redirection events to construct fault tolerant sockets.

2.1 Failover Semantics

We describe the semantics of redirection in RedSocks through a simple example. As shown in Figure 2(a), a communication channel exists between two nodes, A and B. An error at B generates an implicit redirection event that changes the "B" endpoint of the channel to C (see Fig-

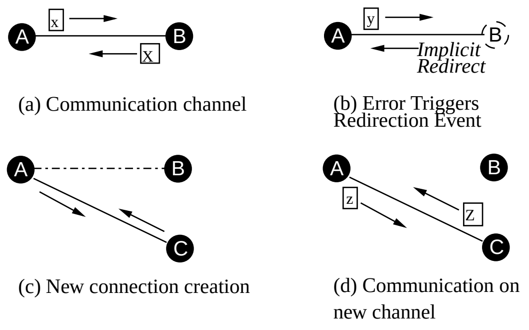
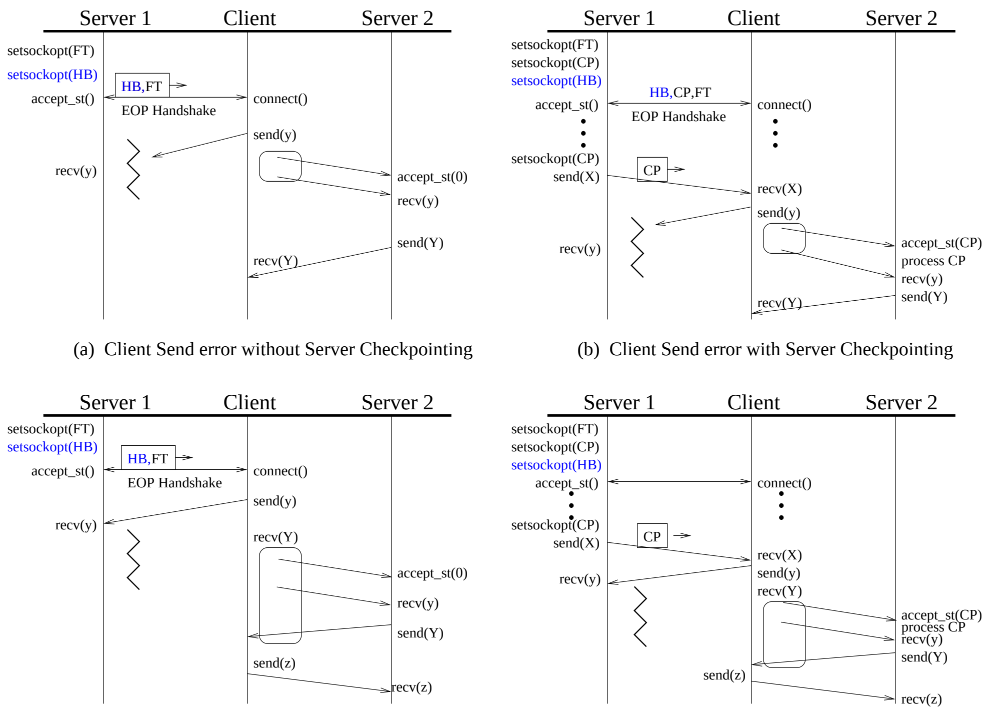

A
B
x
x
(a) Communication channel
y
Implicit Redirect
(b) Error Triggers Redirection Event
C
(c) New connection creation
z
z
(d) Communication on new channel

Figure 2. Failover from endpoint B to C



(a) Client Send error without Server Checkpointing

(b) Client Send error with Server Checkpointing

(c) Client Recv error without Server Checkpointing

(d) Client Recv error with Server Checkpointing

Figure 3. Implicit Redirection Event Timeline

ure 2(b)). C represents a node that may equal A, B or a completely different node. EOP responds to the implicit redirection event by creating a new channel between A and C, synchronizing application and communication state, and removing the channel between A and B as seen in Figure 2(c). In Figure 2, a lower-case letter in a box represents a request and the corresponding upper-case letter represents the response to that request.

2.2 Implicit Redirection Events

Implicit redirection events are generated by errors on the communication channel and serve as triggers for failover. Detecting these errors is not trivial and is explained in Section 3. Implicit redirection is enabled and conditioned via the `setsockopt` system call. We define three new EOP socket options — heartbeat, checkpoint, and failover list. The EOP heartbeat option starts a heartbeat mechanism, at a rate determined by the server, on the client side of the communication channel. The heartbeat mechanism is necessary for detecting host failure or network partitioning, because normal TCP detection of these errors typically takes well over twenty minutes. The EOP checkpoint option buffers server application state at the client and is used to update the alternate server to which the client is directed. Alternate servers are provided to the client-side EOP by the EOP failover list option. No failover can occur until this option is set by the server. The last component that enables implicit redirection is that the client side EOP layer buffers the last data sent and can resend this data, if necessary, to handle in-flight data. Our form of failover has several appealing properties. First, our mechanism depends only on communication channel feedback, i.e. TCP errors, and eliminates the need for 3rd party health monitors prevalent in other solutions. Second, the mechanism directly handles state transfer and synchronization. Other solutions, see Section 5, often involve dedicated backup servers or complex communication channel taps that constantly record packet traffic. Finally, by buffering data at the client and being able to redirect across a WAN, ours is the only solution that handles catastrophic failures.

Figure 3 provides a system call level view of host interactions for the different implicit redirection cases. In Figure 3, a lowercase letter denotes a request, an uppercase letter denotes a reply, a rounded box represents EOP processing that is transparent to the client, and 0 indicates a null parameter. In addition, we use FT to denote the fault tolerant list option, HB to denote the heartbeat option, and CP to denote the checkpointing option. The HB option is shaded in the figure. If used, then the jagged line in the figure represents both process failures and network partitioning. Otherwise, it only represents process failures.

Figure 3(a) gives the case where the server is not using the checkpoint option and the error is detected during a `send` system call at the client. Server 1 must set the FT socket option before accepting the connection. If the server does not, the connection will be vulnerable to errors until the option is set. The server can also set the FT socket option at any time in order to update the list of alternate servers. This list, along with all other option settings, is passed to the client during the EOP handshake (see Section 3). The client-side EOP is notified of a communication channel error via TCP error codes or the heartbeat option. It then picks an alternate server from the failover list, creates a new connection to this server, forwards the previous request y to Server 2, and then returns control to the client. Normal communication ensues.

Figure 3(b) only differs from Figure 3(a) in that checkpointing is used. Server 1 updates the checkpointed state as needed. This state usually reflects application layer state associated with serving requests on the communication channel, such as a file name and offset. The checkpointed state is buffered at the client as part of the session layer communication state and overwritten on each update. The error detection and redirection at the client occur as before, except that the checkpointed data is forwarded to Server 2 arriving in an `accept_st` parameter. The client again forwards the last request. Server 2 sends a reply to this request and continues to serve any additional requests.

Figures 3(c) and 3(d) are the `recv` error counterparts of Figures 3(a) and 3(b), respectively. While the client is blocked in the `recv` system call, EOP detects the error, creates a communication channel to an alternate server, forwards the previous request, forwards any checkpointed data, and then continues to wait for a reply. The client has the capability to set an EOP socket option that will return an EREDO error instead of automatically buffering and forwarding the previous request at failover. This allows for more control when the client is EOP-aware.

The servers need to carefully handle data synchronization in persistent storage with implicit redirection. For example, in Figure 3(d) Server 1 could have accomplished various levels of processing on the client's request before failing. If this processing entails updating a persistent store, Server 2 might repeat updates to this store causing data corruption. If such updates are possible, then the servers could commit updates after the request/response transactions completes, have a rollback mechanism invoked when servers fail, or have a 3rd party mechanism to record the partial processing done that Server 2 can query.

2.3 Usage

RedSocks are designed to be highly flexible and adapt to a multitude of uses, so our description of their usage should be viewed as a guideline and not a hard and fast rule. Trans-

```
s=socket(   SOCK_EOP_STREAM );
HB = setHB();   CP = setCP();   FT = setFT();
setsockopt(s,EOP,EOP_HB,HB);
setsockopt(s,EOP,EOP_CP,CP);
setsockopt(s,EOP,EOP_FT,FT);
ns = accept_st(s,cpState);
if( process(cpState) )
     send(ns,response)
begin loop
     recv(ns,request);
     response = process(request);
     if( cpUpdateNeeded() )
          CP = newCP();
          setsockopt(ns,EOP,EOP_CP,CP);
     send(ns,response);
end loop
```

(a) Server

```
s = socket( SOCK_EOP_STREAM );

begin loop
     send(s,request);
     recv(s,response);
end loop
```

(b) Client

Figure 4. RedSocks Sample Psuedo-Code for Implicit Redirection.

parency is a very important aspect of usability in this area of research. While we show client-aware usage of RedSocks, we actually provide various levels of transparency, including full client transparency, and discuss this in detail in Section 2.4.

For implicit redirection event generation, the server conditions the socket with the system call `setsockopt(level,option,value)`. We created a new level, the EOP level, and a set of EOP options for use with this system call. The options are EOP_HB, EOP_FTLIST, and EOP_CP. The EOP socket options are used to turn on a heartbeat mechanism used to expose network partition errors, to send a list of alternate servers the client uses during failover, and to allow a server to checkpoint application layer state that is transferred to an alternate server if failover occurs. These options were described in Section 2.2 and there implementation is discussed in Section 3.

Figure 4 gives client/server psuedo-code for using RedSocks failover capability. Bold font indicates RedSocks specific changes. The server sets all necessary EOP options that are transferred to the client at send time or via the EOP handshake. The server actions parallel those of a server handling explicit redirection, except that instead of knowing exactly when to send application state, the server sends checkpointing state when necessary. If desired, the client can set an option to expose failover (not shown in the Figure 4). If it does, then the client must handle the EREDO error in Figure 4(b).

2.4 Transparency

Total client-side transparency is achievable by providing a simple wrapper to the BSD Socket API. This might be needed for enhancing legacy applications when it cannot be guaranteed that server interactions will not generate an EREDO error. For a system using BSD Sockets, a wrapper solution would need to intercept all `recv`, `send`, or `socket` system calls. The library would need to buffer all messages sent by the client between receives and be able to resend those messages in the case of an EREDO error. Also, the library needs to substitute SOCK_EOP_STREAM for SOCK_STREAM at socket creation. All other interactions and errors are passed through. Such a library provides full transparency to the client.

Not all applications are suited for complete client-side transparency. For example clients may want to be informed of redirection events in order to enforce different security polices or to adapt their requests for performance reasons. RedSocks includes a socket option that can expose redirection by returning an EREDIR error code whenever redirection occurs, whether explicit or implicit. EREDO takes precedence over EREDIR as it also indicates that a client action is required.

A proxy solution can also be used to transparently provide RedSocks functionality for a wide set of clients or server systems. For example, a web proxy can participate in redirections with any enabled server in the Internet for web content retrieval while maintaining standard communications with its clients.

3 Implementation

There are several different, complimentary strategies for implementing RedSocks. You can use a library, proxy, session layer solution, or RedSocks can be directly incorporated into the application. In our implementation, we chose a session layer solution and include a TCP option to discover the protocol, so that it can be deployed incrementally. Thus, machines can begin aggressively incorporating RedSocks and its functionality can be incrementally used as peers follow suit. Protocol discovery is not an issue for new applications which directly incorporate the protocol and know *a priori* that all participants incorporate it as well – HTTP is an example of this situation. Library solutions will need to implement a protocol discovery mechanism like the one presented in [13].

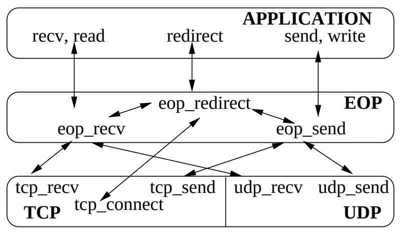


Figure 5. The architecture for EOP and RedSocks

3.1 Environment

We built our fault tolerant sockets in the Linux 2.4.16 kernel, which implements BSD sockets that conform to version 4.4BSD. All code is written in C and the kernel was compiled with egcs-2.91.66.

3.2 Architecture

The fault tolerant socket architecture, shown in Figure 5, is an extension of the architecture we used for RedSocks [5]. We discuss the EOP and transport layer changes necessary to implement our fault tolerant sockets.

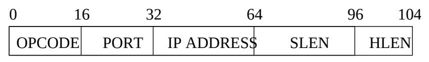


Figure 6. The new EOP Header

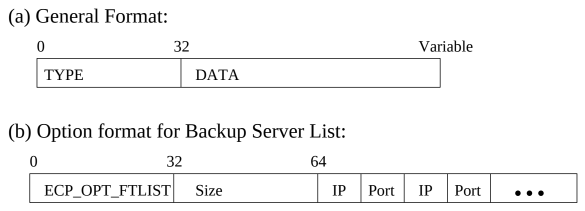


Figure 7. EOP option formats

3.2.1 Endpoint Operation Protocol

We modified the EOP header to allow options. We use the header options to send the backup server list, heartbeat rate, and checkpointed application data necessary for our fault tolerant socket implementation. The new EOP header is given in Figure 6. The additional field *hlen* is a 8-bit field that provides the length of the EOP header plus the options. Figure 7 illustrates the general EOP option format and the specific format for sending a backup server list.

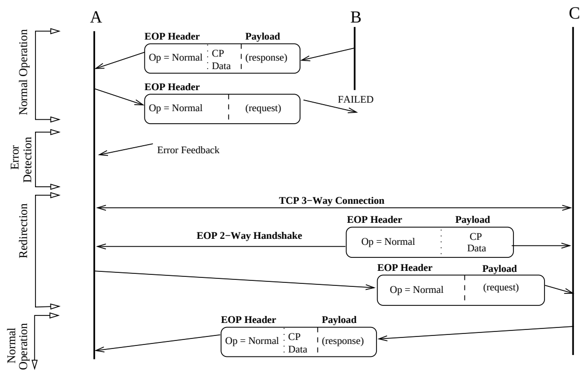


Figure 8. Timeline for EOP packet flow for implicit redirection with a response and application state

Implicit Redirection Time Flow Diagram Figure 8 gives the time flow diagram for an implicit redirection event that includes checkpointed state. B periodically checkpoints its application data during normal communication with A. When B fails, EOP receives communication channel feedback indicating an error which triggers redirection. First, EOP chooses an alternate server from the list provided at connection time[1], connects to it, provides the checkpointed data during the 2-way EOP handshake, and forwards the previous request. Communication between A and C proceeds normally.

Endpoint Failure Detection On the client, the failure of the server-side endpoint is only detectable through errors reported by the transport layer. The EOP layer must capture all relevant error codes to know when to invoke the failover mechanism. The failure mechanism is essentially the RedSocks *redirect* function called by error condition handling code on the client instead of a specific "redirect" EOP header sent by the server and using a backup server list instead of a target in the EOP header.

Table 1 depicts five different failure scenarios. It provides the type of error that is reported to the application, TCP's response, and the fault tolerant action that should be taken for both process and machine failure for each scenario. For network partition failures that separate intermediate routers, we rely on feedback from ICMP messages.

Scenario I describes the errors that can occur during the first connection attempt by the client. If an error occurs at this time, there is no failover action to be taken by the client. In other words, fault tolerance does not apply until a connection is established.

Scenario II, which represents failures that occur during a client send, has two subclasses defined for process failure. When the server process fails, the server-side kernel responds with a partial close of its end of the socket. The client still sends its message and the receipt of that message generates a TCP reset, denoted RST in the table, from the server-side TCP. If the RST arrives before the client calls *recv*, then we get case (a) in the table: the error is detected and the failover operation should be invoked. If the RST arrives after the client calls *recv*, then the client can only detect the sending of FIN which causes the *recv* call to return with zero bytes read. From the point-of-view of the client, this seems like a normal EOF occurring in the socket stream. To distinguish between these cases, we added an ENDTOKEN that the server sends, in the EOP header, on a normal close. If client receives a FIN but no ENDTOKEN, then the client knows it encountered a failure and should fail over. When the server-side machine fails during scenario II, the client encounters an extremely long ACK timeout [2]. This delay is not transparent to the client and another solution is required to detect this condition. Our solution is have EOP periodically send heartbeat messages.

In scenario III, process failures are immediately detected and the client fails over. With machine failure, scenario III poses a more serious problem. No error conditions are generated and the client will block on the *recv* system call indefinitely. Again, we require a heartbeat mechanism to handle this situation.

Last, scenario IV is the connection that occurs during the failover operation. When errors are detected the client tries the next server in the backup server list. If every server on the list is down, then the socket terminates and reports the error to the client.

3.3 Deployability

We enhanced our fault tolerant socket implementation by adding a negotiation mechanism that allows both end-points to discover socket functionality. At connection-time, each end-point checks its peer and discovers whether EOP is supported and adjusts accordingly. We accomplished this by adding a new TCP option that sends the "conditioning" of the socket during the TCP 3-way handshake needed to form a connection. If no option arrives during the handshake, then the remote end is not enabled with our enhanced socket features. Since TCP ignores all unknown options, there is no negative impact in sending this option to systems that do not support it.

Figure 9 shows the format for this option. The *kind* field identifies the option. The *length* field equals the total length of the option, which is four bytes in this case. The last two bytes comprise a bitmask indicating what features are sup-

[1]The use of the option that provides a alternate server list is not shown in the diagram

[2]Twenty three minutes for our Linux 2.4.16 test

FAILURE SCENARIO	PROCESS FAILURE			MACHINE FAILURE		
	FEEDBACK			FEEDBACK		
	ERROR TYPE	TCP RESPONSE	ACTION	ERROR TYPE	TCP RESPONSE	ACTION
I. Initial Connection	connect: refused connection	server: RST	none	connect: no route to host	client: SYN 3 times	none
IIa. Send	recv: broken pipe	server: FIN and RST before recv	failover	recv: no route to host	ACK Timeout = 23 minutes	(heartbeats) failover
IIb. Send	recv: returns 0 no error	server: FIN and RST after recv	no end token?, failover			
III. Recv (partial data)	recv: connection reset by peer	both sent RST	failover	none	none	(heartbeats) failover
IV. Failover Connection	same as I	same as I	failover	same as I	same as I	failover

Table 1. TCP error messages and responses for given server failure scenarios with desired fault tolerant socket behavior.

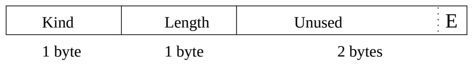


Figure 9. A new TCP option format for socket functionality discovery.

ported. At this time, we only have one feature — EOP enabled (E).

4 Experiments

RedSocks provide the capability to recover from communication channel failures. As failures are the exception and not the rule, the performance of normal communication transfers are extremely important. When a failure does occur, however, a slow recovery may be as unacceptable as the error itself. As we will show next, RedSocks exhibits low overhead in both cases. Last, we provide a proof-of-concept experiment by incorporating RedSocks into the Apache web server.

4.1 Environment

We ran our experiments on different combinations of five 400Mhz dual-processor Pentium III machines where each has 256MB of RAM, two 18GB hard drives, and a 100Mbps Ethernet network card. These machines are connected via a 100Mbps hub. Each machine ran a RedSocks-enhanced Linux 2.4.16 kernel.

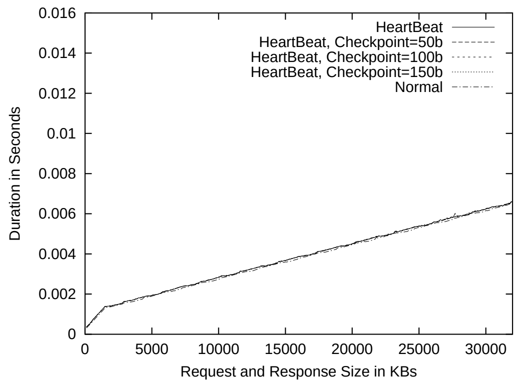
4.2 Fault Tolerance Overhead

We first measured the overhead of using our fault tolerant sockets when no failure occurs using two of the machines described in Section 4.1. We measured the normal communication exchange between our custom client/server applications with our heartbeat mechanism enabled for different sizes of data to checkpoint. The checkpointing scheme we employ is to checkpoint application state before every `send` system call. This simulates the situation where every request is for a file to download which typically occurs in ftp and web servers. We show the client latency for reply/request exhanges, varying in size from 100 bytes to 32K, in Figure 10(a). To serve as a basis for comparison, we include the results for communication with heartbeats and checkpointing turned off; the line labeled "Normal" represents these results.

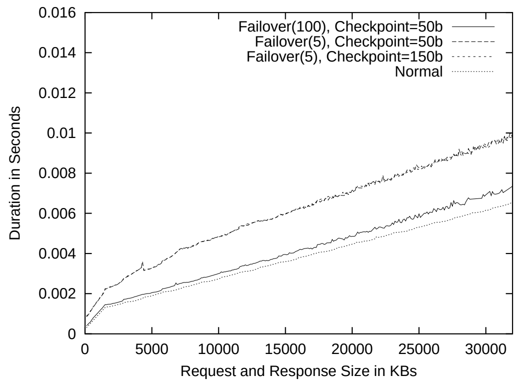
As Figure 10(a) clearly shows, heartbeats and checkpointing add minimal overhead. A careful inspection of the graph reveals that the "Normal" line skims beneath the rest of the cases employing heartbeats and checkpointing.

For our failover overhead measurements, we use three of the 400Mhz dual-processor Pentium III machines described in Section 4.1. The server provides the client with a list of alternate servers to failover to when the client connects via the EOP handshake. At some random point in the communication with the client, the server aborts the connection. When the client-side EOP detects the failure, it selects an alternate server from the failover list provided by the original server and opens a connection to it. During the EOP handshake, the client-side EOP provides the alternate server-side EOP with the checkpointed data, the number of bytes read since the checkpointed data arrived, and any data leftover in the client's send buffer. This information is passed up to the alternate server via the `accept_st` system call (see Section 2.3). The alternate server parses this data and continues to server the request. The entire transaction is transparent to the client.

Figure 10(b) shows our results for this scenario for different simulated file sizes and checkpoint data size. Again,



(a) Communication Overhead



(b) Failover

Figure 10. Failover measurements

we include the results of communication without failure, heartbeats or checkpointing for comparison; these results are represented by the line labeled “Normal” in the graph. We wanted to study the impact of failure on both large and small files. To simulate a large file transfer, we performed 100 hundred `send`/`recv` exhanges of the sizes indicated by the x-axis of Figure 10(b). Simarly, we simulated a small file with 5 such exchanges. For a large files, the time associated with failover is amortized across 100 `send`/`recv` exhanges and thus exhibits less overhead than that experienced by small files.

4.3 Fault Tolerant Apache

We present our experiment setup in Figure 11. Each client simulates multiple browsers by forking children that connect to our modified Apache server and issue a fixed sequence of requests for a file. These requests are idempotent HTTP GET operations for a 2MB file. In this experiment,

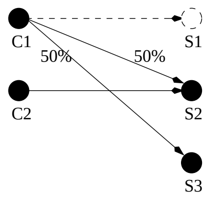


Figure 11. The Experimental Setup

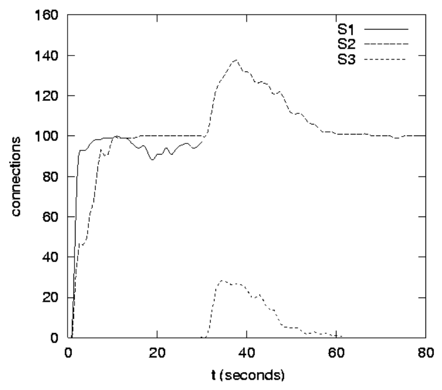


Figure 12. Server Connection Load Across a Server Failure

each client forked 100 children. When each of `C1`’s children initially connect to `S1`, they receive a permutation of $< S2, S3 >$ as their backup list. In this experiment, `C2` simply generates load on `S3`. `C1` initially sends its requests to `S1` until `S1` fails, when 50% of `C1`’s active connections fail over to `S2` while the other 50% fail over to `S3`. The servers sample the number of open connections they are servicing every second. Our goal here was to demonstrate the feasibility and correctness of our fault tolerant socket implementation so we do not address the redirection of new calls to *connect* by the client application. Therefore, after `S1` fails, `C1`’s subsequent calls to *connect* fail.

Modifying Apache to support our fault tolerant sockets required only two trivial changes. First, we added code to parse a backup server list from a configuration file and convert that list into an array of *sock_addr_in* structs. Second, we added a call to *setsockopt* to associate a permutation of the backup list with each connection. These changes required only 25 lines of code. Clients that use fault tolerant sockets to communicate with our modified Apache were then able to recover from communication failures of idempotent HTTP requests, simply by reissuing the request.

We ran our experiment for 100 seconds and killed Apache on `S1` after 29 s had elapsed. Figure 12 depicts the number of concurrent connections open on each server during the experiment. Until `S1` failed, `S1` and `S3` were servicing between 90 and 100 connection, while `S2` was idle. After S1 failed, half of its active connections fail over to S3, whose connection load jumps to approximately 135 connections, and half to S2. Since C1's subsequent new connections fail after S1, as explained above, both S2 and S3 return to their prior connection loads after those connections active when S1 failed are closed.

5 Related Work

Fault Tolerant RedSocks is the only application-independent solution that handles catastrophic failures. We briefly describe application-independent approaches that handle other server-side failures and place our work within the context of these approaches.

The Stream Control Transport Protocol [12] proposes the notion of *multi-homing*, in which an end-point can be associated with multiple IP-addresses. Upon a network failure, the protocol arranges for data to be sent over to an alternate network path to the same server endpoint. Our mechanism also handles network failure by allowing failover to an alternate network path between the same server endpoint or a different one.

HydraNet-FT [10] uses a redirector for detecting failures and re-mapping an existing connection to a secondary server. The redirector is aware of all replicas, and redirects each client request to a primary and all its replicas. The system, thus, uses a primary-backup scheme. Our approach does not require dedicated replicas.

Alvisi et. al [1] describe a fault tolerant TCP (FT-TCP) in which a failed TCP connection can be restored to a specific communication state after a server recovers from failure. FT-TCP is implemented by a wrapper that checkpoints connection state and data at a separate logger. During the recovery process, the wrapper and the loggers interact to bring the application to a specific state. Our recovery process does not require servers to replay all previous communication events to rebuild communication and application state.

Snoeren et. al [11] propose a connection failover mechanism to provide fault tolerance for a collection of Internet servers. In this approach, each connection is associated with a set of LAN-connected servers. The servers periodically synchronize their connection states. Upon failure, the servers in the group contact the client, which then selects one of the servers to resume connection. Because the server resumes communication, the solution is not compatible with NATs or Firewalls. Our approach differs from [11] in that our notion of connection failover is client-centric. Once a connection fails, a client endpoint determines (through a server list) the server to which it should re-connect. This makes our approach compatible with both NATs and firewalls.

6 Conclusion

Fault tolerant RedSocks bring reliability to Internet Applications. Ours is the first, general-purpose solution that handles all server-side failures, including catastrophic failures, in a scalable manner. In future work, we plan to incorporate connection redirection, instream redirection, and fault tolerance to provide a single solution to Internet application scalability and reliability requirements.

References

[1] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *the Proceedings of IEEE INFOCOM 2001*, pages 329–338, Anchorage, AK, USA, April 2001.

[2] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the Ninth International World Wide Web Conference*, volume 33(1–6) of *Computer Networks*, pages 1–16, Amsterdam, The Netherlands, 15–19 May 2000.

[3] T. Brisco. DNS support for load balancing. RFC 1794, Rutgers University, April 1995.

[4] Scaling the Internet web servers. `http://www.cisco.com/warp/public/751/lodir/scale.wp.htm`, 1997.

[5] M. Haungs, R. Pandey, E. Barr, and J. F. Barnes. A fast connection-time redirection mechanism for internet application scalability. In *International Conference on High Performance Computing*, page TBA, Bangalore, India, December 2002.

[6] Michael Haungs. Providing network programming primitives for internet application construction. Technical Report PhD Dissertation, University of California, Davis, September 2002.

[7] K. Li and B. Moon. Distributed cooperative apache web server. In *In Proceedings of the 10th International World Wide Web Conference*, pages 555–564, Hong Kong, May 2001. ACM Press.

[8] M. Luo and C. Yang. Constructing zero-loss web services. In *IEEE Infocom 2001*, Anchorage, Alaska, April 2001. IEEE.

[9] Michael Reene. The customer expectation gap. 2002 Enterpulse Survey, www.enterpulse.com.

[10] G. Shenoy, S.K. Satapati, and R. Bettati. Hydranet-ft: Network support for dependable services. In *Proceedings of the 20th IEEE International Conference on Distributed Computing System*, pages 699–706, Taipai, April 2000. IEEE.

[11] A. Snoeren, D. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 221–232, San Francisco, CA, March 2001.

[12] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. *Stream Control Transmission Protoco*. IETF, rfc 2960 edition, Oct 2001. `http://www.ietf.org/rfc/rfc2960.txt`.

[13] V.C. Zandy and B.P. Miller. Reliable sockets. http://citeseer.nj.nec.com/zandy01reliable.html.