Transparent TCP Connection Failover

R. R. Koch, S. Hortikar, L. E. Moser, P. M. Melliar-Smith
Eternal Systems, Inc.
5290 Overpass Road, Santa Barbara, CA 93111
and
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract

This paper describes a system that enables the failover of a TCP server endpoint in a manner that is transparent to the clients and to the server applications. The failover can occur at any time during the lifetime of a connection. The failover is achieved by modifying the server's TCP/IP stack. No modifications are required to the client's TCP/IP stack, the client application or the server application. The system supports active or semi-active replication of the server.

1. Introduction

To render a service highly available or fault tolerant, the server application must run in a special environment. One approach that is taken runs the application on specialized hardware. Such systems remain operational even if some of their components fail; unfortunately, such systems have a high cost. An alternative approach uses standard computers in a cluster, which is popular because of the lower hardware cost. If a node fails, the application is transferred to a different node in the cluster. In practice, clusters range in size from two to several hundred nodes.

In a cluster solution, the failover of an application from one node to another is not entirely transparent. Although techniques like IP takeover [4] and IP aliasing [14] allow a backup server to take over the identity of a failed server, connections that are established at the time of the failover can no longer be maintained. Several solutions have been proposed to tackle this problem, but all of them require modifications of network edge or core routers, the client application or the client's TCP/IP protocol stack. Those approaches suffer from the drawback that the network and the client typically belong to different organizations than that of the server.

In this paper we describe a system that allows failover of a TCP (Transmission Control Protocol [9]) server endpoint in a transparent manner. The failover can occur at any time during the lifetime of the connection. The failover is achieved by modifying the server's TCP/IP stack. No modifications to the client's TCP/IP stack, the client application or the server application are required.

Although this paper focuses on two-way replicated systems, the proposed solution is not limited to two-way replication. Higher degrees of replication can be achieved by daisy-chaining multiple backup servers. The description of such systems is beyond the scope of this paper.

When failing over a TCP server endpoint from a primary server to a secondary server, the server application must be present on both hosts. We assume that the server application process is actively replicated. With active replication, the server application runs on both hosts. Both server processes accept connections, handle requests and generate replies. They both go through the same state transitions. The server process must behave deterministically on a per connection basis. By that we mean that when a client connects to a server and issues a request, it will receive a particular reply.

An on-line store is an example of a deterministic service. Unless two customers compete for the last remaining item, each client will get a well-defined response to a browse or purchase request — independent of the fact that the server implementation uses an independent thread per client.

In TCP connection establishment, one side (the TCP server) listens for incoming connection requests, while the other side (the TCP client) connects to the server. The approach discussed in this paper allows for the replicated server application to act as a TCP server (e.g., a replicated Web server that accepts connection requests from unreplicated clients) or as a TCP client (e.g., a replicated Web server that connects to an unreplicated back-end database).

First, we give the requirements for a TCP failover solution. Then we describe the workings of the failover mechanism for a single TCP connection in the fault-free case. Next we examine its behavior in the case that the server fails. Finally, we describe the connection establishment and

termination procedures. Reintegration of failed servers is beyond the scope of this paper.

The TCP failover mechanisms reside in the primary and secondary servers' network stack between the TCP layer and the IP layer. Throughout the paper, we refer to this sublayer as the *bridge*.

2. Maintaining the Correct State of a TCP Connection

The TCP layer resides above the IP (Internet Protocol) layer. TCP accepts messages from the user application and divides the message into TCP segments. The TCP segments are passed to the IP layer, where they are packed into IP datagrams. The routers that reside between the client computer and the server computers work at the IP layer and, therefore, have no knowledge of TCP.

To perform a TCP connection endpoint failover from a primary server to a secondary server that is transparent to the client, four requirements must be satisfied:

1. IP datagrams that the client sends to the primary server must be redirected to the secondary server. A solution is given in [4].

2. The secondary server must have a copy of all TCP segments sent by the client that the primary server has acknowledged. The primary server must not acknowledge a client's TCP segment until it has received an acknowledgment of that segment from the secondary server.

3. The secondary server must have a copy of all TCP datagrams sent by the primary server that have not been acknowledged by the client. If the client acknowledges a server TCP segment, the primary server and the secondary server must each receive the acknowledgment and remove the segment from its buffers.

4. The secondary server must synchronize its TCP sequence numbers with the sequence numbers used by the primary server. The order of the sequence numbers must not be violated in case of a failover. The client will reset the connection if it detects a violation in the order of the sequence numbers.

In addition, the secondary server must respect the Maximum Segment Size (MSS) [10] and the maximum window size that were negotiated between the primary server and the client at connection establishment.

To detect the failure of a server process or server host, the system employs a fault detector.

3. Connection Management in the Fault-Free Case

We consider a client application running on host C that communicates to a replicated server application with primary server P and secondary server S via a TCP failover connection, as shown in Figure 1.

The client application sends a request to the server by passing the request message to the TCP layer. The TCP layer packs the data into TCP segments. Each segment contains a unique sequence number. Next the TCP segment is passed to the IP layer, which packs the TCP segment into an IP datagram. The IP datagram header contains the IP address of the sender (source) host and the IP address of the receiving (destination) host. In this particular case, the source address is the IP address of the client a_c. The destination address of the datagram is the IP address of the primary server a_p.

3.1. Secondary Server

The secondary server, whose network interface runs in promiscuous mode, receives all of the client's datagrams. The secondary server bridge discards all datagrams that do not contain a TCP segment or that are not addressed to P. For all other datagrams, the bridge replaces the original destination field with the address a_s of the secondary server and passes the datagram to the TCP layer. TCP assumes that C sent this segment directly to S. The TCP layer extracts the original client request and passes it to the server application.

After the secondary server application has processed the client's request, it generates a reply. The secondary server TCP layer generates one or more TCP segments that contain the reply and passes them to the secondary server bridge.

If the secondary server bridge receives a segment that is addressed to the client C, it replaces the destination address field of the segment with the address of P. Thus, all TCP segments intended for the client are diverted to P. The original destination address of the segment is included in the segment as a TCP header option.

Modifying the TCP header of the segment requires recalculation of the TCP checksum. Note that it is not necessary to recompute the checksum from scratch. Instead, we subtract the original bytes from the checksum, and add the new bytes to the checksum.

3.2. Primary Server

When receiving the datagram on the network, the IP layer of P delivers the content of the datagram to the TCP layer, which then extracts the original client request and passes it to the server application.

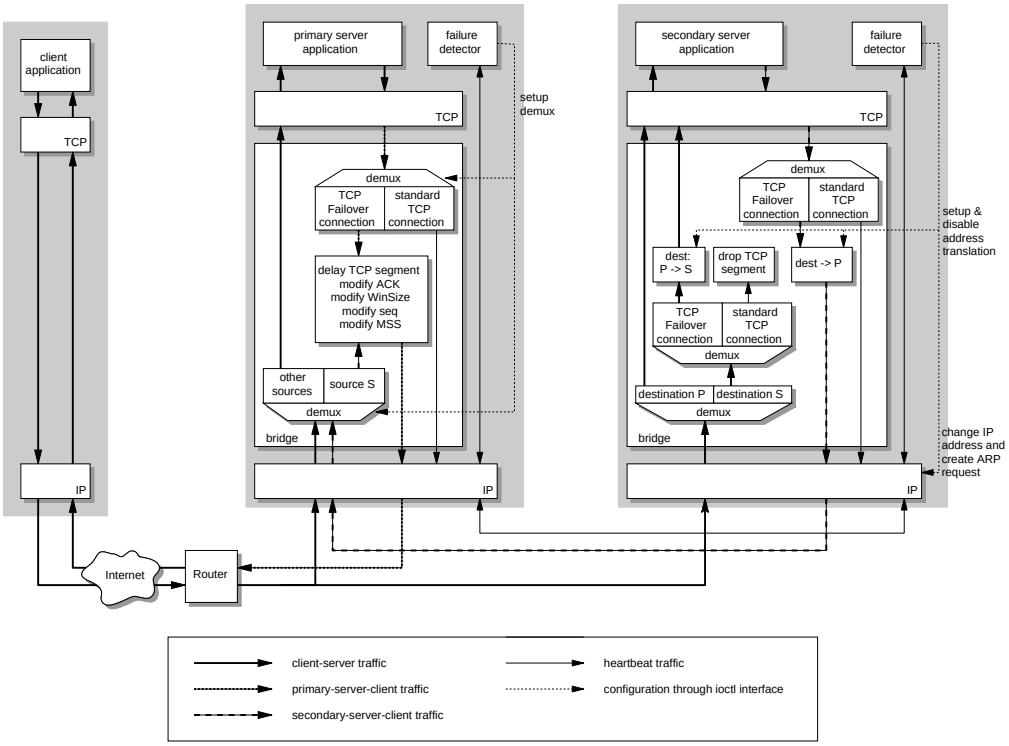

Figure 1. Structure used to achieve the transparent failover of a TCP connection endpoint.

After the primary server application processed the client's request, it generates a reply. If the server applications behave deterministically, both replies are identical.

The TCP layers pack the replies into TCP segments. Note that, although the application replies are identical, the TCP layers might not generate two identical sets of TCP segments. Due to flow control, one of the server's TCP layer might split the reply into multiple TCP segments, whereas the other server's TCP layer might pack the entire reply into a single segment.

When obtaining segments from the TCP layer, the primary server bridge must not send the segment directly. Instead, it puts the payload in the primary server output queue and waits until it receives corresponding data from S. The primary server bridge must not send any data to the client until it has received the data from S and its own TCP layer.

When the primary server bridge receives the TCP segment sent by S, it matches the segment's payload against the content of the primary server output queue. The bridge constructs a new segment that contains all of the matching payload bytes. The remaining bytes of the original segment are enqueued in the secondary server output queue.

The new segment carries the address of the primary server P in the source field and the address of the client C in the destination field. The acknowledgment field contains either the acknowledgment sequence number of the last segment the bridge has received from P or S, whichever is smaller. The same procedure is used to fill the window size field of the new segment.

Choosing the smaller of the two acknowledgments guarantees that both servers have successfully received all of the clients data up to the sequence number of the forwarded acknowledgment. Choosing the smaller of the two window sizes adapts the client's send rate to the slower of the two servers and, thus, reduces the risk of message loss.

3.3. Synchronizing Sequence Numbers

To establish a new connection, P and S choose starting sequence numbers $\text{seq}_{P,init}$ and $\text{seq}_{S,init}$. The primary server bridge calculates the the sequence number offset Δ_{seq} as the difference between the two initial sequence numbers: Δ_{seq} = $\text{seq}_{P,init} - \text{seq}_{S,init}$. See Section 7 for more details.

The primary server bridge synchronizes to the sequence numbers generated by the secondary server. The primary server bridge receives all segments generated by the secondary server. The synchronization is achieved without any additional communication between the two servers.

To compare the sequence numbers of segments sent by the secondary server S and the primary server P, the primary

server bridge subtracts Δ_{seq} from the sequence numbers of all segments it receives from the primary server's TCP layer.

3.4. Constructing new TCP Segments

Figure 2 illustrates the primary server bridge building TCP segments. On the left side of the figure, we see the primary server bridge receiving a segment from the primary server's TCP layer. The segment contains the payload bytes 51 to 54. After subtracting Δ_{seq}, which we assume to be 30, from the sequence number, the payload bytes are enqueued in the primary server output queue. After that, the bridge receives a segment sent by the secondary server that carries the payload bytes 23 to 26. The bridge finds and removes matching payload bytes 23 and 24 in the primary server output queue, and creates a new TCP segment. The remaining bytes 25 and 26 are enqueued in the secondary server output queue.

If the bridge obtains a TCP segment from a server but cannot build a TCP segment because the queue of the other server does not contain any matching payload, it compares the minimum of P's and S's most recent acknowledgments with the acknowledgment of the previous TCP segment it built. If the former is greater than the latter, the bridge constructs a TCP segment with no payload. This prevents a deadlock in case the server applications do not send any data to the client. In this case, TCP must send empty segments to acknowledge the client segments.

Being a duplex connection, TCP tries to piggyback acknowledgments of a data stream to the segments of the stream that goes in the other direction. If no data is sent in the other direction, TCP creates a delayed acknowledgment. A delayed acknowledgment is a TCP segment that carries no user payload. If the bridge receives such a segment, it updates the ACK and WinSize fields of the sender and compares the new ACK value with the ACK of the last segment that it sent to the client. If the former is greater than the latter, the bridge constructs a TCP segment with no payload.

4. Loss of Messages

If a segment m is dropped in TCP, two things happen at the receiver. First, the receiver will not acknowledge m or any later segments that the sender of m sends. After the sender's retransmission timer expires, it retransmits m. Second, the receiver will not receive the acknowledgment ack_k that the sender attached to m. ack_k acknowledges the receiver's segment k. If the sender does not send additional segments that acknowledge k, the receiver's retransmission timer expires, and the receiver retransmits k.

The TCP failover extension must be able to handle message loss. Message loss can occur at the following places:

- The primary server does not receive a client segment m. The TCP layer of the primary server P does not acknowledge m. Consequently, the primary server bridge does not acknowledge m. The client C retransmits m after C's retransmission timer expires.

  Message m might carry an acknowledgment ack_k for a segment k that the server sent. Because the primary server does not receive ack_k, it retransmits k. By comparing k's sequence number with the last sequence number it sent, the primary server bridge recognizes that k is a retransmission. It, therefore, does not enqueue k, but sends k immediately. This is necessary because the bridge receives only a single copy of k.

- The secondary server drops the client segment although the primary server receives it. This case is similar to the case described above.

- A client segment is lost on its way to the servers. If neither S nor P has received the client's segment m, neither has received ack_k and, therefore, both retransmit k. In this case, the primary server bridge sends k twice.

- The secondary server's segment is dropped by the primary server. If a segment m sent by the secondary server S is not received by the primary server bridge, the bridge is not sending any more segments to the client C. Consequently, C will never acknowledge m or any later server segments, and both servers will retransmit m. Assume that the bridge receives S's retransmission first. As soon as it receives S's copy of m, it sends m to the client C. When it receives P's copy, the bridge recognizes this copy as a retransmission and sends it again. In case the bridge receives P's copy first, it finds m in P's queue and discards the second copy of m. As soon as the bridge receives S's copy of m, it sends m to the client.

- The primary server's segment is lost on its way to the client. If a segment m is dropped on its way from the primary server bridge to the client C, the client C will not acknowledge m. Consequently, both servers will retransmit m after their retransmission timer expires. Again, the primary server bridge will send two copies of m to C.

5. Failure of the Primary Server

If the fault detector detects that the primary server failed, the secondary server performs the following tasks:

1. Tell the secondary server bridge to stop sending TCP segments to the IP layer that are addressed to the client.

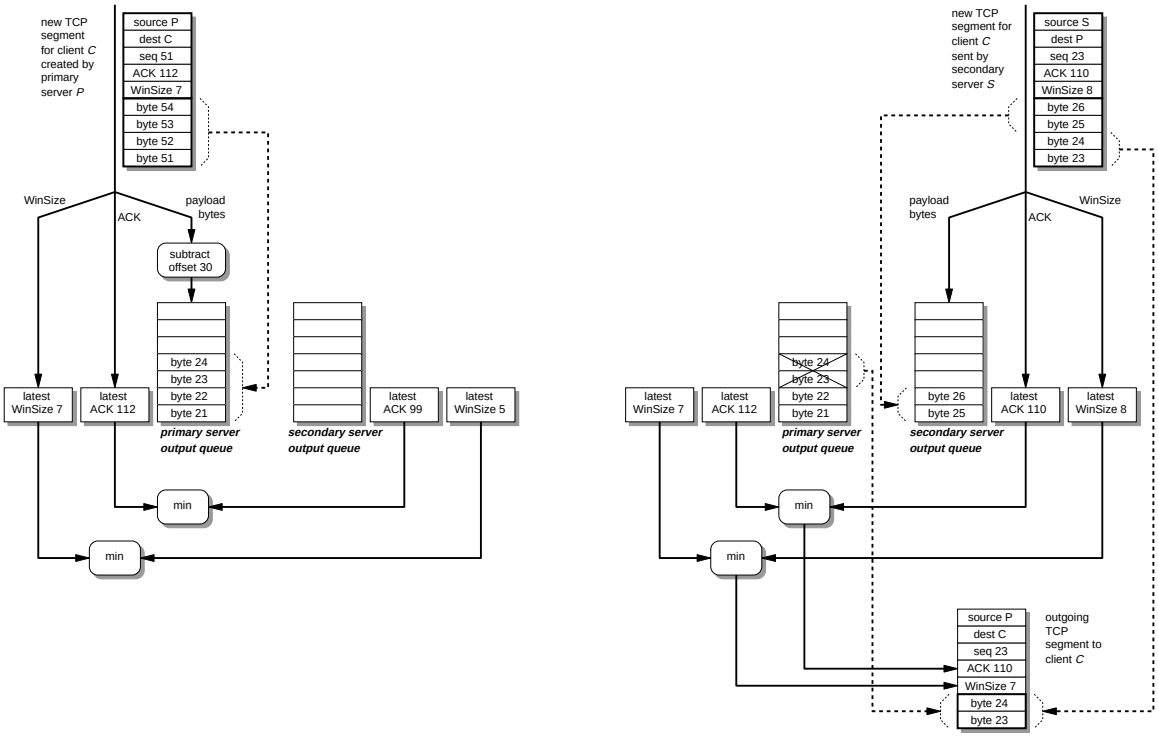

Figure 2. Primary server bridge constructs TCP segments.

2. Disable the promiscuous receive mode of the network interface.

3. Disable the a_p-to-a_s address translation of the destination field for incoming TCP segments.

4. Disable the a_c-to-a_p address translation of the destination field for outgoing TCP segments.

5. Take over the IP address of the primary server.

After the change of IP address is completed, the bridge resumes sending TCP segments.

If the primary server fails, it is guaranteed that the secondary server has received all TCP segments that the primary server has acknowledged. If the secondary server has received additional segments, it acknowledges them. However, those acknowledgments are sent to the primary server as long as the bridge has not been reconfigured.

We let T be the time interval from the time of the failure of the primary server to the time the router updates its Address Resolution Protocol (ARP) table as a response to the ARP request that the secondary server sent. None of the TCP segments that the secondary server sent during T reaches the client, which has two effects. First, the client will not acknowledge any of those segments, and the secondary server will retransmit those segments. Second, the client will not receive acknowledgments for any segments it sent after the primary server failed, and the client retransmits those segments periodically.

The secondary server can receive data from the client until the promiscuous receive mode of its network interface is disabled. The secondary server's TCP layer discards TCP segments that the client retransmitted if it has received a copy of those segments.

After the completion of the IP takeover, the secondary server sends its acknowledgments directly to the client. The secondary server acknowledges only those segments it has received. The secondary server does not receive a client's segments if it dropped them before the IP takeover occurred, or if the router forwarded the segments between the time that the secondary server disabled the promiscuous receive mode and the router updated its ARP table. The client will retransmit any unacknowledged segments. Because the client has not received an acknowledgment from the primary server, the client must have a copy of those segments.

During the reconfiguration of the secondary server bridge, neither the sequence number counter nor the ACK sequence number nor the window size needs to be changed. All TCP segments that have been sent to the client contain sequence numbers that match the sequence numbers of the segments that the secondary server generated. The TCP segments that the primary server sent to the client carried the

smaller of the ACKs and window sizes that were advertised by the TCP layers of the primary server and the secondary server.

Once the IP takeover is completed, the secondary server disables all functions of its bridge and behaves like any standard TCP server.

6. Failure of the Secondary Server

If the fault detector detects a failure of the secondary server, the primary server performs the following tasks:

1. Remove all payload data from the primary server output queue, place the data into a newly created TCP segment (or multiple segments, if necessary), and send the segment to the client.
2. Disable the demultiplexer for incoming IP datagrams. Route all incoming TCP segments directly to the TCP layer.
3. Disable the delay of TCP segments that the primary server created. Do not modify the acknowledgment field or the window size of those segments. However, continue to subtract the offset Δ_{seq} from the sequence number field of all outgoing TCP segments that are addressed to the client.

After the completion of the recovery from the failure of the secondary server, all TCP segments that the primary server sent to the client contain the acknowledgment ack_P and window size win_P that the primary server's TCP layer chose.

During normal operation, all segments that the primary server bridge sends to the client carry sequence numbers that the secondary server S assigned. The bridge adjusts all sequence numbers assigned by the TCP layer of the primary server P by subtracting Δ_{seq}. In case the secondary server fails, the bridge of the primary server must not discontinue to compensate the offset because the client's TCP layer is synchronized to the sequence numbers that the secondary server generated.

7. Connection Establishment

The primary and secondary server bridges must be able to distinguish between TCP failover connections, which are serviced by the replicated server, and ordinary TCP connections. This distinction must be made for each segment that passes through the primary and secondary server bridges.

We implemented two methods for specifying whether a TCP connection is a TCP failover connection. In the first method, the socket interface was augmented to allow the application program to set the TCP failover option for each streaming socket it opens. This scheme is flexible and elegant, but requires modification of the application source code.

In the second method, the user can enable the TCP failover option for a set of port numbers. All connections that use one of those ports are treated as TCP failover connections. The user must specify the same set of ports on the primary server host and the secondary server host.

7.1. Client-Initiated Connection Establishment

Establishing a TCP connection involves a three-way handshake. First, the endpoint that requests the connection (client) sends a TCP segment that has the synchronization flag set (SYN segment) to the listening endpoint (server). The SYN segment specifies a server port and contains the client's initial sequence number. If the server wants to accept the connection, it sends back a SYN segment that acknowledges the client's SYN segment. The server's segment contains the server's initial sequence number and an acknowledgment for the client's SYN segment. For the third step, the client acknowledges the server's SYN segment. The connection is then established, and either side can send TCP segments.

When the client sends its initial SYN segment to establish a TCP failover connection, both primary and secondary server receive the SYN segment. The primary server bridge passes the SYN segment to the TCP layer. When the TCP layer accepts the connection request, it sends a SYN segment in return. On receiving this segment from the TCP layer, the primary server bridge creates the primary and secondary server output queues, and enqueues the segment. At the same time, the bridge stores the sequence number $\text{seq}_{P,init}$ of that segment to be able to perform the sequence number offset calculation.

No special action is necessary at the secondary server. The secondary server bridge performs the address translation for the incoming and outgoing SYN segments based on the network ID of the client endpoint's IP address.

When the primary server bridge receives the SYN segment that the secondary server's TCP layer created, it calculates the sequence number offset Δ_{seq} by subtracting the sequence number of that segment $\text{seq}_{S,init}$ from $\text{seq}_{P,init}$. Then the primary bridge constructs the SYN segment to be sent to the client. The MSS field of that segment is set to the minimum of the Maximum Segment Size (MSS) fields contained in the SYN segments that the TCP layers of the primary and secondary servers created. By sending the TCP segment, the primary server bridge completes its initialization procedure.

The client TCP layer completes the three-way handshake by sending an acknowledgment for the server's SYN segment. The primary server bridge and the secondary server

bridge handle the acknowledgment segment in the same way as all future incoming segments.

The primary server must maintain a primary server output queue and a secondary server output queue for every active TCP connection. A TCP connection is uniquely identified by the 4-tuple (client IP address, client TCP port number, primary server IP address, primary server TCP port number).

7.2. Server-Initiated Connection Establishment

The primary server P and the secondary server S initiate the establishment of a TCP connection to an unreplicated server T (e.g., back-end database server in a multi-tier system) by sending a SYN segment. Assuming that the application running on P and S is deterministic, both P's TCP layer and S's TCP layer generate a SYN. When it receives the first SYN segment, P's bridge creates the output queues and enqueues the segment. When the bridge of one server receives the other server's SYN segment, it calculates the sequence number offset, creates a SYN segment and sends it to T. When the TCP layer of T accepts the connection request, it sends a SYN segment in return. The primary server bridge and the secondary server bridge handle the acknowledgment segment in the same way as all future incoming segments. The servers complete the three-way handshake by sending an acknowledgment for the client's SYN segment.

8. Connection Termination

Terminating a TCP connection involves a four-way handshake. TCP requires that each direction of the connection is shut down independently of the other. To terminate one direction of a TCP connection, the sending endpoint must send a TCP segment with the FIN flag set. Either side can initiate the connection termination process. The other endpoint acknowledges the FIN segment. The connection is then in a half-closed state, in which the endpoint that has not sent the FIN is still allowed to send data. The other endpoint must acknowledge all incoming segments, but is not allowed to send data. The half-closed state prevails until the side that remained active sends a FIN. As soon as the other side acknowledges the FIN, the connection is closed.

As in connection establishment, only the primary server bridge is actively involved in connection termination. The primary server bridge remains active as long as the connection is not fully closed. In a half-closed state, the primary server bridge must merge the segments generated by the primary and backup servers. As long as the client-to-server side remains open, the primary server acknowledges client segments only if the secondary server has acknowledged those segments. Otherwise, the secondary server might not have segments that the primary acknowledged when a failover occurs. As long as the server-to-client side remains open, the primary server must not send any segments to the client before it receives identical segments from the secondary server.

If the primary server bridge receives the first FIN from the client, it marks the TCP connection closed by the client. As soon as it sends the FIN that the server generated, the bridge marks the connection as fully closed. It then waits for the client's acknowledgment of the server's FIN and deletes all internal data structures that were allocated for the connection. If the secondary server S does not receive the client's ACK for the FIN segment, S retransmits it. When the bridge receives a FIN that S sent after the bridge removed all internal data structures associated with the connection, it creates an ACK and sends it back to S.

If the TCP layers of the primary and secondary servers terminate the connection, the primary server bridge internally marks the TCP connection as closed by the servers. As soon as the bridge receives the FIN sent by the client C, it marks the connection as fully closed. It then waits for the servers' acknowledgment of the client's FIN. The bridge deletes all internal data structures that were allocated for the connection after it sent the segment that contains the ACK of the client's FIN. If the client does not receive the servers' ACK, it retransmits the FIN. When the primary server bridge receives a FIN sent by the client C after it removed all internal data structures associated with the connection, it creates an ACK and sends the ACK back to C.

9. Measurements

To measure the performance of the TCP Failover protocol, we conducted a number of experiments. The TCP Failover protocol was implemented in the FreeBSD 4.4Lite kernel, which ran on 566MHz Pentium III Celeron PCs. The client computer was a 1GHz Pentium III PC running Mandrake 7.2 Linux, which ran the 2.2.17 kernel. The PCs were connected using 100Mbit/s Ethernet.

All measurements were done from the viewpoint of a client application that communicates with a redundant server using TCP Failover.

We first measured the connection time from a client application to a redundant server and compared those numbers to the standard TCP connection time. We made sure that the MAC addresses of all nodes were present in the ARP caches. If the MAC addresses are not cached, the client and the router must run the ARP protocol, which adds about 300μs to the connection setup time. The time for ARP affects standard TCP and TCP Failover in the same way.

The median connection setup time for standard TCP is 294μs with a maximum of 603μs; the median connection time for TCP Failover is 505μs with a maximum of 1193μs.

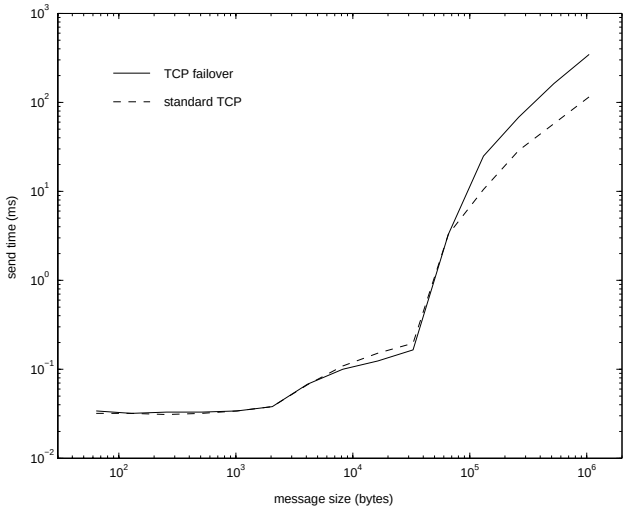

Figure 3. Client-to-server data transfer.

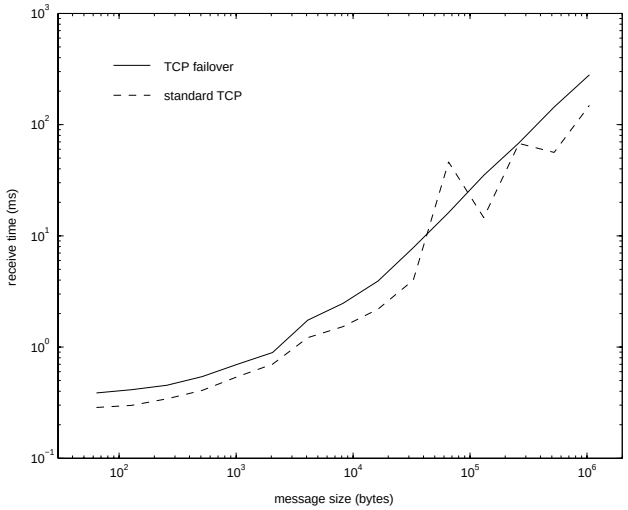

Figure 4. Server-to-client data transfer.

Next we measured the send and receive times of messages of different length. Figure 3 shows the median time it takes an application to send a message to an unreplicated server using standard TCP, and to a replicated server using TCP Failover. The message length varied from 64 bytes to 1 MByte.

It can be seen that the send time for messages up to 32 KBytes does not increase at the same rate as the send times for larger messages. This is due to the 64 KByte TCP send buffer. The send call returns when the application has passed the last byte to the stack, not when the last byte has been put on the wire. The effect of the send buffer decreases with increasing message size.

We obtained similar results for server-to-client data transfer. In this case, the client application sends a 4-byte message to the server, and the server sends a reply message back to the client. Figure 4 shows the time that elapsed between the client starting to send the 4-byte message, and the client receiving the last byte of the servers' reply. The size of the reply messages varies from 64 bytes to 1MByte. The non-linearity in the standard TCP measurement is caused by collisions on the Ethernet. The probability of acknowledgments colliding with data packets varies with message size.

Figure 5 compares send and receive rates between standard TCP and TCP Failover. The rates were measured by having a client send and receive data streams of 100 MBytes.

We choose the File Transfer Protocol (FTP) to test TCP Failover with a real-world application. The File Transfer Protocol (FTP) allows a client to upload and download files from a remote size. The remote site runs an FTP server, which listens on a well-known port (port 21). An FTP client opens an ephemeral port (a port chosen by the operating system) and connects to the server's FTP port. This connection is used to exchange control data. After the server has verified that the client is permitted to access the server's file system, the client opens a server socket with an ephemeral port and informs the server of the chosen port number.

Every time the client initiates a data transfer (`get` or `put`), it sends a request to the server. The server opens a client socket on port 20 (FTP data) and connects to the client. The server and the client exchange the file content via the data connection. Once the transfer has completed, both sides terminate the data connection.

We connected an FTP client to the replicated FTP server via a wide-area network and transfered files of different sizes. Figure 6 lists median send and receive rates as indicated by the FTP client.

As these results illustrate, measurements over a wide-area network are highly dependent on competing traffic and on packet loss rates and, thus, vary widely.

	standard TCP connection	TCP Failover connection
send rate	7833.70KB/s	5835.80KB/s
receive rate	8707.88KB/s	3510.03KB/s

Figure 5. Comparison of send and receive rates for long data streams.

file size [KB/s]	get file		put file	
	standard TCP	TCP failover	standard TCP	TCP failover
0.2	8.75	8.75	512.38	536.05
1.3	59.03	59.03	2033.76	2036.87
18.2	90.41	70.74	3846.13	3890.42
144.9	156.80	138.35	219.52	200.31
1738.1	176.03	171.72	168.07	176.63

Figure 6. FTP send and receive rates in KBytes/s.

10. Related Work

TCP splicing [13] is a technique that is used to improve performance and scalability of application-level gateways. Clients establish TCP connections to a dispatcher application. The dispatcher chooses an appropriate server to handle a client connection. Then the dispatcher modifies the TCP stack of the dispatcher host to forward all TCP packets of that connection directly to the selected server. No further involvement of the dispatcher is necessary until the connection is terminated.

TCP splicing requires that all traffic flows through the dispatcher. TCP handoff [3] removes the dispatcher by letting the client connect directly to one of the servers. If the initial server decides that another server is better suited to handle the connection, it transfers the TCP connection state to an alternative server. TCP handoff requires a special front-end layer-4 switch that routes the packets to the appropriate server.

Snoeren and Balakrishnan [12] describe a TCP migration scheme that is transparent to the client application but requires modification to both the client and server TCP layer. A change in the network infrastructure (e.g., Internet routers, underlying protocols) is not required. The migration of the connection can be initiated by the client or any of the servers. The replicated servers can be geographically distributed. At any point in time, only one server is connected to the client. Multicasting or forwarding of the client's data is not possible.

Sultan, Srinivasan, Iyer and Iftode [15] propose M-TCP, a TCP connection migration scheme that moves a server's TCP endpoint to a different location. In addition to migrating the TCP endpoint, M-TCP moves a limited amount of application state and synchronizes the application and the TCP layer. M-TCP requires the support of both the client and server TCP layer. The migration is initiated by the client. During the migration process, both servers are required to be operational, which renders this approach unsuitable for fault tolerance.

Shenoy, Satapati and Bettati [11] propose a fault-tolerant extension of the HydraNet infrastructure to replace a single server with a group of replicated servers. Their approach does not require any modification of the client's TCP layer. Instead, all IP packets sent by the client to a certain IP address and port number are multicast to a set of replicated servers, which can be geographically distributed. For this scheme to work, all traffic must go through a special redirector, which resides on an Internet router. To maintain consistency between all server replicas, the system supports atomic multicasting. The forwarding service is not restricted to TCP, but can accommodate any transport protocol that is based on IP.

Alvisi, Bressoud, El-Khashab, Marzullo and Zagorodnov [2] describe a system in which all client-server TCP communication is intercepted and logged at a backup computer. When the server fails, the server application is restarted and all stack activity is replayed. The backup node performs an IP takeover and takes over the role of the server node for the remaining lifetime of the connection. No modifications are required to the client TCP stack, the client application or the server application. To operate properly, the backup node must be operational before the connection between the client and the server is established. Although the failover happens transparently to the client, the failover time can be significant due to the replay of the entire history of the connection.

Orgiyan and Fetzer [8] describe a system that replicates a server application in a semi-active manner. TCP server endpoints are replicated. Similar to the TCP Failover approach, their approach puts the network interface of the secondary server into promiscuous mode. The system employs a leader/follower protocol to avoid inconsistent behavior caused by non-determinism. The snooping of network traffic reduces the overhead of the leader/follower protocol communication. Their approach requires the modification of the server application and the system libraries of the server and the client hosts. It is not clear whether the system is always able to maintain ongoing TCP connections in case of a node failure. If the secondary server drops a TCP segment that the primary server has acknowledged and then the primary server host fails, the segment cannot be recovered, the connection must be abandoned and the client must reestablish the connection.

Fetzer and Mishra [5] propose a system that allows the transparent replication of servers that communicate to unreplicated clients via TCP. This approach appears to be very similar to TCP failover. The secondary server taps into connections that are established between the primary server and the client by using a promiscuous receive mode. The client remains unmodified.

SwiFT [6, 7] provides fault tolerance for user applications, including modules for error detection and recovery,

checkpointing, event logging and replay, communication error recovery and IP packet rerouting. The latter is achieved by providing a single IP image for a cluster of computers. Addressing within the cluster is done by MAC addresses. All traffic from clients is sent to a dispatcher, which forwards the packets to one of the servers. The clients must run the SwiFT client software to reestablish TCP connections in case the server fails.

Aghdaie and Tamir [1] describe a system to replicate Web servers. The basic concept of their solution is similar to the TCP failover approach. To avoid changes to the server operating system, the authors implemented their scheme in user space by using IP sockets. The server application is passively replicated. The backup proxy logs client requests and server replies. The drawback of this scheme is the poor performance resulting from the context switches and protocol stack traversals that are needed for an implementation entirely in user space.

11. Conclusion

We have described TCP Failover, a protocol that enables the failover of a TCP server endpoint in a manner that is transparent to the clients and to the server application. If a fault occurs, TCP Failover migrates the TCP server endpoint from a primary server to a backup server. The failover is achieved by modifying the server's TCP/IP stack. No modifications are required to the client's TCP/IP stack, the client application or the server application. The overhead is reasonable, given that the approach is completely transparent to both the clients and the server application.

References

[1] N. Aghdaie and Y. Tamir, "Client-transparent fault-tolerant Web service," *Proceedings of the IEEE International Conference on Performance, Computing, and Communications* , Phoenix, AZ (April 2001), pp. 209–216.

[2] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo and D. Zagorodnov, "Wrapping server-side TCP to mask connection failures," *Proceedings of INFOCOM 2001*, Anchorage, AL (April 2001), pp. 329–337.

[3] M. Aron, D. Sanders, P. Druschel and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based network servers," *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA (June 2000), pp. 323–336.

[4] A. Bhide, E. N. Elnozahy and S. P. Morgan, "A highly available network file server," *Proceedings of the 1991 USENIX Winter Conference*, Dallas, TX (January 1991), pp. 199–205.

[5] C. Fetzer and S. Mishra, "Transparent TCP/IP based replication," *Proceedings of the IEEE International Symposium on Fault-tolerant Computing*, Madison, WI (June 1999).

[6] H. Y. Huang and C. Kintala, "Software implemented fault tolerance," *Proceedings of the IEEE Fault Tolerant Computing Symposium*, Toulouse, France (June 1993), pp. 2–10.

[7] Y. Huang, P. E. Chung, C. Kintala, C.-Y. Wang and D.-R. Liang, "NT-SwiFT: Software implemented fault tolerance on Windows NT," *Proceedings of the USENIX Windows NT Symposium*, Seattle, WA (August 1998).

[8] M. Orgiyan and C. Fetzer, "Tapping TCP streams." *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA (October 2001), pp. 278–289.

[9] J. B. Postel, "Transmission Control Protocol," RFC 793 (September 1981).

[10] J. B. Postel, "TCP maximum segment size and related topics," RFC 879 (November 1983).

[11] G. Shenoy, S. K. Satapati and R. Bettati, "HydraNet-FT: Network support for dependable services," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan (April 2000), pp. 699–706.

[12] A. C. Snoeren, D. G. Andersen and H. Balakrishnan, "Fine-grained failover using connection migration," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA (March 2001), pp. 221–232.

[13] O. Spatscheck, J. S. Hansen, J. H. Hartman and L. L. Peterson, "Optimizing TCP forwarder performance," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2 (April 2000), pp. 146–157.

[14] W. R. Stevens, Unix Network Programming, vol. 1, 2nd edition, Prentice-Hall, Upper Saddle River, NJ (1998).

[15] F. Sultan, K. Srinivasan, D. Iyer and L. Iftode, "Migratory TCP: Connection migration for service continuity in the Internet," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vienna, Austria (July 2002), pp. 469–470.