

# Efficient TCP Connection Failover in Web Server Clusters

Ronghua Zhang, Tarek F. Abdelzaher, John A. Stankovic

Department of Computer Science

University of Virginia

Charlottesville, VA 22904

{rz5b,zaher,stankovic}@cs.virginia.edu

**Abstract**—Web clusters continue to be widely used by large enterprises and organizations to host online services. Providing services without interruption is critical to the revenue and perceived image of both hosts and content providers. Therefore, server node failure and recovery should be invisible to the clients. Most of the existing fault-tolerance schemes simply stop dispatching future client requests to the failed server. They do not recover those connections handled by the node at the time of failure, which makes the failure visible to some clients. Making the failure transparent requires both application-layer and transport-layer mechanisms. While atomic application-layer primary-backup failover schemes have been addressed at length in previous literature, a transport-layer scheme is necessary in order to make them invisible to the clients. In this paper we describe a transparent TCP connection failover mechanism. Besides transparency, our solution is also highly efficient, and does not need any dedicated hardware support.

## I. INTRODUCTION

Today, an increasing number of organizations and enterprises are providing their services online. They have a large stake in the reliability of the server clusters hosting their web sites. A web site failure may be significant in both short-term revenue loss and long-term reputation damage. A famous example is e-bay [1], whose 10-hour service outage in 1999 cost it at least \$5 million dollars along with other side effects.

A typical server cluster consists of a front-end dispatcher and many back-end servers, any of which can fail at any time. Depending on the functionality of the failed node, the strategies to deal with its failure may be quite different. Since the failure of the front-end will lead to a complete service interruption, its failure should be avoided as much as possible. A common practice is to add a hot standby dispatcher, which keeps monitoring the primary dispatcher and will take over its responsibility if it crashes. In contrast, when a back-end server fails, the dispatcher simply excludes it from the working set, thus avoiding dispatching any future requests to it until it comes back. However, the connections being processed when the crash occurs are lost. Consequently, some unlucky clients are exposed to the failure, and have to resubmit their requests. They are given an impression of an unreliable web site, and may not return, which is the last thing service providers are willing to see.

To completely hide the server failure from users, support from the transport layer as well as the application is necessary. The transport layer transparently migrates those

ongoing connections from the failed node to a healthy one. After the connections are migrated, the user's requests are reissued to the application. Many applications have *execute-exactly-once* transaction semantics. Therefore, the application should then decide whether or not to execute those reissued requests in order to preserve the transaction semantics. In this paper, we only focus on the transport layer effort, i.e., how to transparently migrate connections. How to preserve the transaction semantics depends on the application, thus is beyond the scope of this paper.

Transparent connection migration is challenging since the solution must satisfy the following conditions in order to become applicable to real systems and be accepted by service providers:

- 1) Compatibility. Considering the extremely large number of Internet users, the solution should not require any modifications to the client-side software including the operating system. For example, the connection migration should not require modifications to the client-side TCP, and should not need cooperation from the client.
- 2) Low cost. Since the server cluster usually has a large number of back-end servers, such techniques as active backup or dedicated network connections among the servers are too costly to be feasible. For the same reason, connection migration should not rely upon any support from special or dedicated hardware.
- 3) Low overhead. The system throughput should not tangibly degrade after this technique is applied. Under normal operating conditions (i.e., when no failures occur), the clients should not experience any extra delay.

Many techniques have been proposed in the literature [2]–[10] to address the failover problem. None of them exhibits all three properties. A more detailed discussion of related work is provided in Section V. In this paper, we present a novel solution to the transparent failover problem that satisfies all three aforementioned conditions: (i) it requires only server side (kernel) modification, (ii) back-end servers back each other up in a p2p fashion and no extra hardware is needed, and (iii) the overhead is very small and no delay is introduced under normal operation.

The rest of this paper is organized as follows. We first present the design and implementation details in section II.

Two optimization techniques are described in section III. We report the evaluation results of our prototype in section IV. Related work is presented in section V. Section VI discusses some further implementation issues and future work. The paper concludes with section VII.

## II. DESIGN AND IMPLEMENTATION

A prototype of the failover protocol was designed and implemented on a local testbed. In this section, the basic architecture is presented. Two optimization techniques that reduce the implementation overhead are described in section III.

### A. Overview of Normal Clusters

A web cluster consists of a front-end and several back-ends connected with a LAN. When the front-end receives a new connection request, it will choose an appropriate back-end to handle this connection. Many factors can affect the back-end selection, such as load balancing, or locality awareness. Further incoming packets for this connection are forwarded to this back-end by the front-end. A hash table is maintained by the front-end, mapping the connection to the back-end handling this connection. This table is updated when a new connection request is received or when a connection is torn down. The front-end thus becomes a single point of failure to the whole cluster. Several techniques have already been proposed and implemented to address this issue [11], [12].

### B. Architecture

In a normal cluster, each connection is handled by one back-end, and only that back-end is aware of the existence and state of this connection. In order to achieve connection fault-tolerance in our scheme, each connection is made visible to at least two back-ends: one primary server and one or possibly several backup servers. During normal operation, the connection is processed only by the primary server, incurring small overhead at the backups. Upon failure of the primary, the connection migrates to a backup. The backup is able to resume the connection transparently before the client TCP times out. The backup TCP interface to the server application makes it look as if the client re-issued a request for a failed connection on the primary. Since many online services require *execute-exactly-once* transaction semantics, the application needs to decide whether or not the re-issued request should be executed in order to preserve the transaction semantics. This decision depends on how the application is designed, and is beyond the scope of this paper. Also, if a backup (to an active primary) fails, a new backup is selected.

We organize all back-end servers into a ring. In general, each back-end is a backup server for a fixed number,  $N$ , of predecessor back-ends in the ring. Similarly, those connections assigned to a back-end by the front-end are backed up by a fixed number,  $N$ , of successors. Our current implementation supports only  $N = 1$ . Thus, below, we focus on the single backup configuration. Extensions to support multiple backups are discussed in section VI-A. While such extensions

are simple, we do not believe that there is much need for them in practice, unless one expects correlated failures to affect both the primary and backup machines at once. Such correlated failures, however, may be reduced by preventive techniques such as putting these machines on different power circuits. Hence, the probability of correlated failures of both the primary and backup can be made much lower than single machine failures in today's clusters.

Figure 1 is an example cluster configuration with  $N = 1$ : backend1 is the backup for backend2, backend2 is the backup for backend3 and backend3 is the backup for backend1. For each connection, the front-end will designate a *primary back-end* server to process it. Its successor becomes the *backup back-end* for the connection. Similarly, for each back-end, those connections for which it is the primary server are called *primary connections*, and those connections for which it is the backup server are called *backup connections*.

The front-end forwards each packet to the primary server as well as the backup server(s). Since the primary and backup servers see the same set of packets,<sup>1</sup> the connection state is consistent across them. When a back-end server fails, one of its backups is notified to recover those connections based on the maintained connection state.

In the following, we describe the four key components of the architecture, namely, (1) failure detection, (2) ring maintenance, (3) connection state tracking, and (4) connection failover.

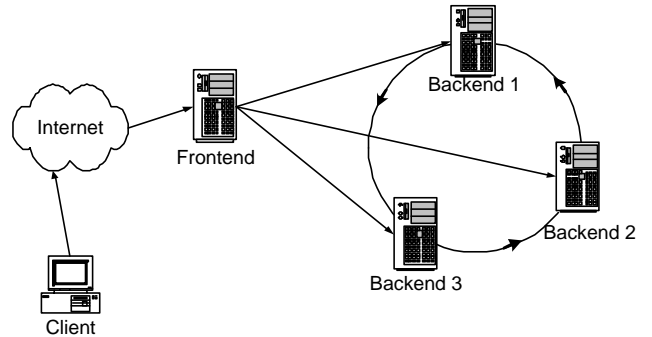


Fig. 1. Ring Configuration Example

### C. Failure Detection

Reliable failure detection is not the focus of this paper. Our technique is independent of the failure detection scheme. Any failure detection scheme can be integrated into our architecture. In the current implementation, a heartbeat scheme is adopted due to its simplicity. Periodically, each back-end sends out a heartbeat message to the front-end. This message can also piggyback some load information on the back-end to improve load balance. If the front-end has not heard from a back-end for some duration, called the *failure detection window*, this back-end is considered to be dead. Consequently,

<sup>1</sup>Packet Loss is not considered here, since it will not happen after the optimization techniques are applied as will be shown in Section III.

one of its backups is notified to take over its connections. Moreover, ring maintenance is invoked to repair the ring structure.

#### D. Ring Maintenance

Ring maintenance is needed in two cases: when a back-end crashes and when a previously crashed back-end comes back.

- A back-end fails. We will use Figure 1 as an example. Let us assume that backend2 fails<sup>2</sup>. Its backup, backend1, will be assigned two responsibilities. First, it is notified to take over the connections previously served by the failed backend2. The state of such connections is already available on backend1. Moreover, since the failed backend2 was itself a backup for backend3 before the crash, backend1 must replace backend2 as a backup to backend3. Notice that backend1 at this point does not have any information about the current connections on backend3. Therefore, backend3 needs to synchronize its current connection state with its new backup, backend1. For each connection, its state variables (only those needed by the backup) and all the application data received so far including out of order packets are transmitted to the new backup. To achieve the latter, the TCP stack of backend3 has to keep all the data packets after they have been delivered to the applications. Otherwise, these packets will not be available to backend1 during the synchronization. Consequently, when the connections are recovered at backend1 in the future, the client requests cannot be replayed to the application. Fortunately, for many online services, most of the incoming TCP packets are pure ACKs. Only a very small number of TCP packets carry application data. Therefore, the overhead of synchronization is not formidable, considering that most of the servers are equipped with very large memory. In our current implementation, to avoid complexity and possible race conditions, the front-end temporarily stops forwarding packets to backend3 until the synchronization is finished. This can be improved by a more complex implementation.
- A back-end revives. When backend2 finally comes back again, the ring structure needs to be adjusted as follows: backend2 becomes the new backup for backend3 and backend1 becomes the backup for backend2. Again, a synchronization procedure may be executed. In addition, the backup information for connections on backend3 kept by backend1 can now be discarded.

#### E. Connection State Tracking

Connection state tracking at the backup servers is achieved via a cooperation between the front-end and the back-end. When the front-end receives a packet from the client, it will forward it to both the primary server and the backup servers. Obviously, only one of the back-ends should process the

<sup>2</sup>Here we assume only one back-end fails at a time. The multiple failure case is handled similarly.

request and send the reply to the client. To achieve this result, we implement a protocol named BTCP (or Backup TCP) on top of IP (Figure 2). When the front-end forwards packets to the backup servers, it will change the protocol field in the IP header to BTCP so that the BTCP layer instead of the TCP layer on the backup servers will process these packets.

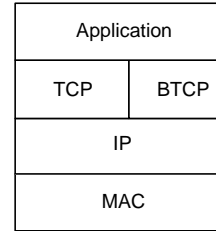


Fig. 2. Protocol Stack on Back-ends

BTCP is basically a 'silent' version of TCP: it processes each incoming packet in a similar way as TCP does, but it never sends back any reply to the client and never interacts with the applications. For each backup connection, it creates a small data structure *backup\_sock* to track its state. Some of the important state variables for each connection are: sequence number of the SYN sent out by the server (ISS), sequence number of the SYN received by the server (IRS), sequence number of the first unacknowledged byte by the client (SND.UNA), window size, and of course, all the application data sent by the client. *Backup\_sock* is created when the connection is established, and destroyed when the connection is torn down.

However, since the backup can only observe incoming traffic, some important information is not available to it directly, which needs to be inferred from other information. Namely:

- *When a TCP connection ceases to exist?* A TCP connection ceases to exist when both sides have sent out a FIN packet and have been acknowledged by the other side. Since the backup can only observe packets sent out by the client, it can only know when the client side has closed the connection (i.e., that the client will not send out any packets except pure ACKs). The time at which the primary server closes the connection is unknown to the backup. Timeout is used to address this problem. If the primary is still sending out data to the client, the client will reply with ACK packets. When the backup has not received any ACK packets over a connection for some duration after it has received a FIN, it can assume that the connection does not exist anymore.

Note that this timeout duration should be properly chosen (according to the application). If the timeout duration is too short, the backup will destroy the state information of some connections prematurely, and not be able to recover them. On the other hand, a longer timeout leads to extra memory usage to keep state for nonexistent connections. What is worse, when connection recovery is required, an attempt to recover those (nonexistent) connections is made including the regeneration of replies by the

applications. When such replies are resent to the client, the client responds with a RST since the connection does not exist anymore. However, lots of resources will have already been wasted during this procedure. The second optimization technique discussed in section III-B alleviates this problem.

- *How to derive server ISS?* Since the initial sequence number on the server side (ISS) is chosen by the primary server and sent out to the client as the reply to the SYN packet, the backup does not know this value directly. Instead, to derive this value, when the backup receives a SYN packet, the sequence number of this packet, IRS, is recorded. When the client sends out an ACK packet to complete the 3-way handshake, the backup first verifies that the sequence number (SEQ) field of this packet is  $IRS + 1$ . If the SEQ value is correct, then we assume this packet is acceptable. Thus its ACK value must be  $ISS + 1$ . In case this ACK value is actually invalid, the primary server sends a RST to abort this connection (remember that the primary and the backup server receive the same packets), so the only possible cost is that of one unnecessary *backup\_sock* for the nonexistent connection. After some timeout, this structure is reclaimed by the kernel since no more packets arrive for this connection.

#### F. Connection Failover

When a back-end is requested to take over from a failed server, the following steps are taken for each backup connection: (1) *backup\_sock* is converted into a normal TCP socket to represent this connection, (2) a connection request is constructed and inserted into the accept queue so that the application on the backup machine can accept this connection, and (3) when the application on the backup accepts the connection, data packets previously received by the primary are delivered to the application.

As we mentioned before, after the requests are re-issued to the application, it is the application's responsibility to decide whether or not those requests will be executed. For the stateless applications, such as serving a static web page, it is safe to resend the requested web page. For those applications with transaction semantics, however, blindly re-executing those requests can cause trouble. For example, the user's credit card may be charged twice. Therefore, the application should be carefully designed to handle this issue.

Note that the primary server may have already sent back some data to the client before the crash. If the backup server sends these data again during the replay, network bandwidth is wasted. However, we know that the amount of data (including the initial SYN) successfully delivered to the client is  $SND.UNA - ISS$ . Hence, when the application begins to send back data, our TCP layer simply discards that amount of application data before actually sending packets with the remaining data to the client. Thus, by maintaining the same logical transport layer connection across a server machine failure, our connection migration scheme, in fact, improves performance over the case where connections on failed servers

are restarted and repeated by the clients. The above mechanisms are transparent to and independent of the application. Obviously, the CPU cycles used by the application to generate data during connection replay are wasted. This overhead may be eliminated with some help from the application or some knowledge of the upper layer protocol as will be discussed in section VI-C.

### III. OPTIMIZATION

It is obvious that the basic implementation introduces extra overhead. More specifically, the front-end needs CPU cycles to forward each packet twice, more network bandwidth consumption is required, and the back-end needs more CPU cycles to process incoming packets on backup connections. In this part, two techniques used to reduce the overhead are presented.

#### A. MAC Layer Multicast

Since the cluster machines are presumably on a LAN, multicast is an efficient way to forward packets. Every back-end is assigned a unique MAC multicast address. Its backups subscribe to the primary's multicast group. When the front-end receives a packet for some connection, it first looks up the hash table to decide the primary server for that connection. Then, the protocol field in the IP header is set to be BTCP, so that the BTCP layer of the back-ends can intercept this packet. Finally, a MAC header is added with the destination address for the packet being the multicast address of the primary server.

The ring maintenance protocol is extended so that the multicast address of the primary back-end is sent to its backup server whenever a primary-backup pair changes. Consequently, each back-end will join two MAC multicast groups: one for itself as primary and one as backup for another primary server.

Note that the primary and backup now receive an identical packet, they need some information to decide whether it is the primary or backup server for this connection. The destination MAC address in the packet is the answer to this question. When BTCP at the back-end receives the packet (recall that the front-end changes the protocol field in the IP header to BTCP), it first examines the destination MAC address in the packet. If the address is its own multicast address, it means it is the primary server for this connection. The packet is redirected to the TCP layer and continues its journey as a normal TCP packet. Otherwise, this back-end is a backup server for this connection and the BTCP logic is invoked to process this packet.

Since the packets are forwarded only once, the CPU consumption of the front-end and the network bandwidth requirement are reduced to those of a normal cluster. An extra benefit is the atomicity of packet forwarding: either both the primary and the backups receive a forwarded packet, or none of them receives it. Therefore, we don't have to worry about inconsistent connection state perceived by the primary and backup servers due to packet loss.

#### B. Selective Multicast

In the basic implementation, the backup server has to process every packet in order to keep track of the connection

state. The incoming packets over a connection can be classified into two categories: (1) packets with connection control information (SYN, RST, FIN, PUSH) or application data, and (2) pure ACK packets. In selective multicast, the front-end only multicasts the first category of packets to both the primary and backup back-ends. It forwards the pure ACK packets only to the primary back-end. Since most of the incoming packets are pure ACKs, this scheme will save a lot of processing time for the backup server.

However, since the backup server does not see ACK packets, it loses track of the sliding window state of the connection. When the connection failover is requested, a probe packet is first sent to the client. The probe packet is an empty TCP packet, and its SEQ field is ISS - 1. This packet is unacceptable to the client,<sup>3</sup> and the client sends out an ACK as a reply [13]. Based on this ACK, the backup server now knows the sliding window state of the connection, and can continue normal connection failover actions as before.

As discussed in section II-E, BTCP at the backup relies on the ACKs to decide the connection existence. If no ACK is ever forwarded to the backup, BTCP at the backup may destroy the connection state prematurely. Therefore, the following forwarding policy is enforced: before the client sends out a FIN, no ACK is ever forwarded to the backup; after the client sends out a FIN, the front-end forwards some of the ACKs to the backup. For example: forward 5% of the ACKs or forward 1 ACK every 2 minutes.

We mentioned earlier that the inability of backups to see primary server FIN packets requires our algorithm to use a timeout to infer connection termination. Should a failure occur before such a timeout expires, resources may be wasted to recover and regenerate replies on previously terminated connections. The probe message can actually alleviate this problem. Since such a message must precede recovery, if a connection no longer exists, the probe message leads to a RST reply from the client, thus terminating the recovery procedure at an early stage, thereby saving resources.

#### IV. EVALUATION

Next, we present an evaluation of our scheme on the implemented system prototype. All the experiments were conducted on a testbed consisting of 15 machines connected with a 100Mbps LAN. The front-end and four back-end machines are slower machines equipped with an AMD K6 450MHz CPU and 256 MB memory. Ten faster machines, each having an AMD Athlon 1GHz CPU with 512 MB memory, are used to generate the workload. This is intentionally arranged in order to stress the cluster and make prominent any overhead introduced by our technique. The front-end and the back-ends all run the Linux kernel version 2.4.19 with our kernel patch applied. The workload generator machines run the Linux kernel version 2.4.20.

<sup>3</sup>We assume that the sequence number space is not reused during the lifetime of the connection. Any reuse we believe, is quite a rare situation.

TABLE I  
REQUEST RATE AND REPLY SIZE USED TO EVALUATE THE OPTIMIZATION  
TECHNIQUES

Reply Size	Request Rate	Bandwidth Requirement
1K Bytes	2000/s	15.6 Mbps
2K Bytes	2000/s	31.2 Mbps
4K Bytes	1800/s	56.3 Mbps
8K Bytes	1200/s	75 Mbps
16K Bytes	600/s	75 Mbps
32K Bytes	300/s	75 Mbps

##### A. Effect of Optimization

The first group of experiments is designed to demonstrate the effectiveness of our optimization techniques by evaluating the three fault-tolerance schemes described; namely, the basic scheme, described in Section II, and the two optimized ones, described in Section III. To make the comparison of overheads easier, we configured the cluster (in this experiment only) in a special way that isolates the overhead of backups more distinctly by letting it be incurred on a separate machine. Hence, in this special arrangement, one of the four back-ends acted solely as the backup for the other three. The front-end did not dispatch any client requests to the backup machine. Therefore, the CPU utilization of this machine is a measure of the cost of maintaining backup information for 3 primary servers. We measure the CPU utilizations on the front-end and the single backup when different optimizations are used. Up to 10 machines running httpperf [14] were used to generate client requests. The experiment was repeated with different request rates and reply sizes to reveal the relationship between the workload and the CPU utilization. For each reply size, we chose the request rate that can saturate the back-end server CPUs or the network. When the reply size is small, from 1K to 4K, back-end server CPUs are the bottleneck, and when the reply size is big, from 8K to 32K, the network bandwidth becomes the bottleneck. The actual request rates and reply sizes used are listed in Table I. The CPU utilization of the front-end and backup server under the different backup techniques we described in the previous sections are compared in Figure 3 and Figure 4, respectively. In the following we refer to these techniques as “forwarding methods” since they differ primarily in how the front-end forwards data to the back-ends.

Observe, from Figure 3, that the front-end CPU utilization follows a similar trend no matter what forwarding method is used. This is because the front-end’s task is to maintain proper state for each connection and forward the incoming packets to the back-end. Therefore, its workload  $U$ , or CPU utilization, is proportional to the connection rate  $R$  and the number of incoming packet  $P$ , i.e.,  $U = aR + bP$ . The values of  $a$  and  $b$  may depend on the forwarding method, but the formula is generally true. Given the traffic listed in Table I, it’s not surprising that the CPU utilization peaks when the reply size is 2 Kbytes.

Also notice that using multicast to forward packets does reduce the front-end CPU utilization substantially. Selective multicast has no further noticeable impact on the front-end

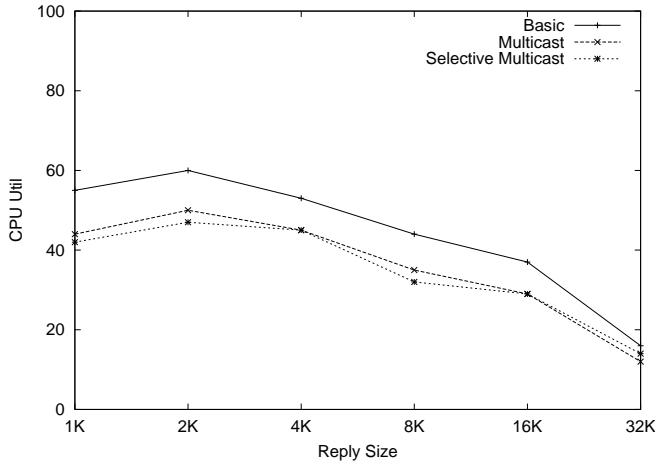


Fig. 3. Front-end CPU Utilization

compared with multicast. These two results are anticipated.

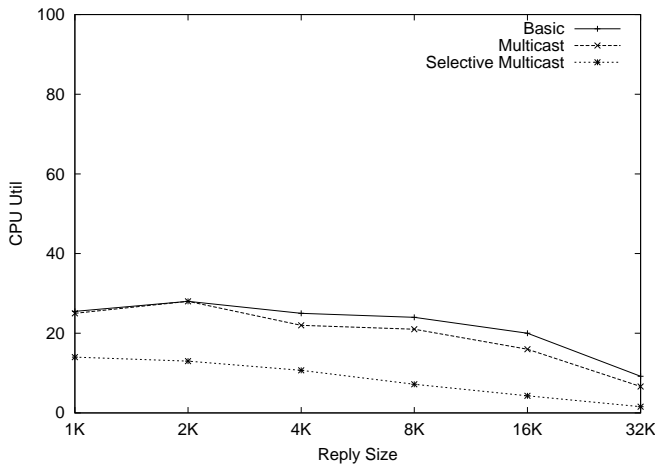


Fig. 4. Backup Server CPU Utilization

Further, two important observations can be made from Figure 4 regarding the utilization of the single backup machine that aggregates the overhead of backing up our three back-end servers. First, selective multicast can reduce the CPU utilization of the backup machine considerably, which is not surprising since the number of packets to be processed is much less in this case. Second, the overhead of selective multicast depends only on the request rate. This is different from the overheads of other approaches which grow with reply size when the request rate is nearly constant as can be appreciated from comparing the first two data points in Figure 4. When the basic forwarding method or simple multicast is used, the backup has to process every packet the clients send to the server. Therefore, it follows a similar pattern to Figure 3 (The shape is still different because the values of  $a$  and  $b$  are different). However, when selective multicast is used, the overhead is decided only by the number of connections. Hence, the CPU utilization decreases as the request rate decreases (see Table I and Figure 4). Finally, observe that the single backup overhead can be approximately estimated

TABLE II

REQUEST RATE AND REPLY SIZE USED TO MEASURE THE OVERHEAD

Reply Size	Request Rate	Bandwidth Requirement
1K Bytes	1200/s	9.4 Mbps
2K Bytes	1200/s	18.8 Mbps
4K Bytes	1100/s	34.4 Mbps
8K Bytes	1000/s	62.5 Mbps
16K Bytes	600/s	75 Mbps
32k Bytes	300/s	75 Mbps

by dividing the curves in Figure 4 by the number of primary servers backed up, which is three. Using this calculation, the maximum overhead imposed by the selective multicast scheme is seen to be around 4-5% per machine.

### B. Operational Overhead

Our previous experiments have successfully demonstrated the efficacy of different optimization techniques and isolated their respective overheads in a special setting. The next step is to quantify the overhead of the best scheme in the actual intended operational setting for our implementation; a ring of back-ends. To study this overhead, it is enough to experiment with a cluster of only 2 back-ends. Each acts as backup for the other one. As before, the experiment was repeated with different request rates and different reply sizes. Again, for a given reply size, the rate was chosen to saturate the back-ends or the network. The actual workload parameters are listed in Table II. Figure 5 compares the CPU utilization of the front-end in the cases where the fault tolerance support is enabled (FT-cluster) and disabled (normal-cluster). The CPU utilization of the back-ends under these two conditions is compared in Figure 6.

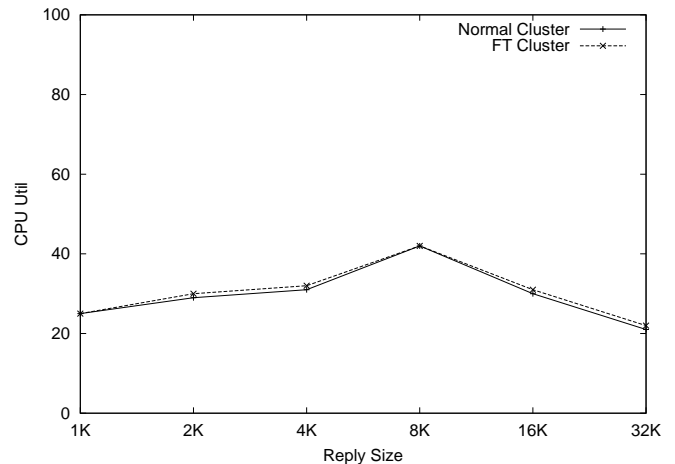


Fig. 5. Front-end CPU Utilization

In both figures, the gap between the two curves is the overhead. As we can see, in terms of CPU utilization, adding fault tolerance support incurs negligible overhead at the front-end, and at most about 6% overhead at the back-end (which is consistent with our previous calculation from Figure 4 for similar loads).

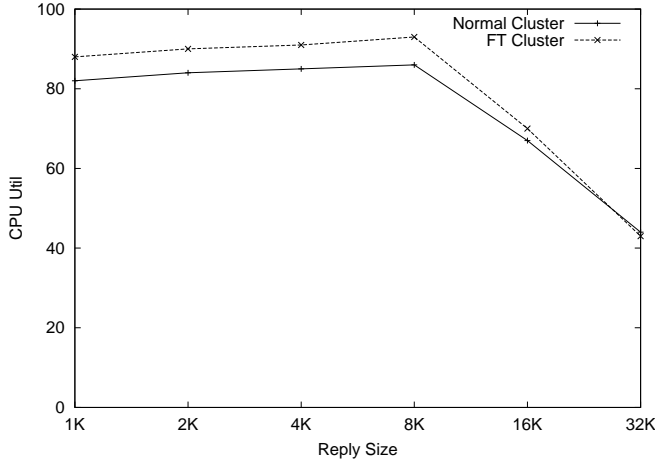


Fig. 6. Back-end CPU Utilization

In terms of memory usage, the front-end needs 8 extra bytes for each connection, 28 extra bytes per service (e.g: FTP, WWW) per back-end, and 60 bytes per back-end to store such information as a multicast address. At the back-end, the normal socket data structure is augmented by 24 bytes, and each backup connection needs around 170 bytes. All these numbers are not big enough to be a concern.

The only extra network traffic resulting from our scheme is the periodic heartbeat messages, which are normal UDP packets with a 4 byte payload. Suppose each back-end sends out 10 heartbeats each second, the network bandwidth usage per back-end is still less than 1K Byte.

In summary, our implementation has low overhead in terms of CPU utilization, memory consumption, and network bandwidth requirement, thus successfully meeting the third requirement mentioned in Section I.

### C. Failover Time

Besides the overhead, another important performance metric is the failover time, i.e., how long it takes to recover an interrupted connection on the backup back-end. Figure 7 depicts the breakdown of the failover time.

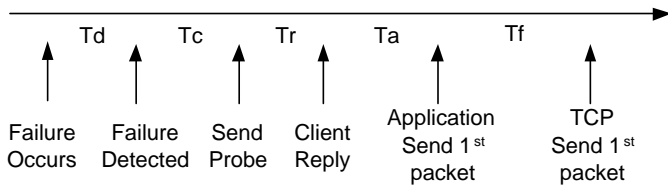


Fig. 7. Failover Time Breakdown

$T_d$  is the failure detection latency, which is determined by the configurable heartbeat period and crash detection window.  $T_c$  is the time it takes the kernel to reconstruct a TCP connection at the backup.  $T_r$  is the duration from the moment the probe message is sent out to the moment a packet is received from the client.  $T_r$  might be shorter than the round-trip time, since the client may start to retransmit a packet

before it receives the probe message.  $T_a$  is the time that the application spends reading and parsing the request. During  $T_f$ , the application is sending out (silently discarded by TCP, though) the portion of the reply which the client has already received. Basically, this is the time it takes to read or generate that portion of data. This time may depend on application-level primary-backup failover schemes that might be enabled on the server. Actually,  $T_r$  and  $T_a$  may overlap with each other, since the sliding window state is not needed until the application begins to send out data.

To give a more realistic estimate of the failover time, we set up a relatively “realistic” environment: the workload was generated from a remote site instead of local machines, and the workload was generated using SURGE [15] to simulate realistic traffic. SURGE [15] is a Web workload generator that can generate traffic matching the empirical models of server file size distribution, file popularity distribution, reply size distribution, temporal locality of request, and think time distribution. The file set used in this experiment consisted of 2000 unique files occupying 56 MBytes disk space. The machine at the remote site simulated 100 clients sending out requests to the cluster. After the experiment had been running for 30 seconds, we unplugged the network cable of a back-end server to simulate the failure. The failure was detected by the front-end after around 0.5s, and another backup server was notified to take over. A total of 98 connections were recovered by the backup within 0.9s. Since the connection recovery only involves some memory copy and one packet transmission, this delay is unexpectedly long. Further investigation has found that the reason for the delay is because we use a kernel thread to recover these connections. This thread has to compete with other user processes for the CPU. Hence, most of the time it is waiting for the CPU instead of performing the recovery work. This problem can be alleviated by giving this thread a higher priority. We have not attempted this modification, as we deem the current delay relatively insignificant to the client anyway.

We repeated the same experiment using a normal cluster (i.e., one with no backup schemes implemented), but without introducing the failure. The response times observed at the remote site during the two experiments were collected. Figure 8 compares the cumulative distributions of the response time throughout the whole experimentation period under these two cases. Observe that these two are very close to each other, which means that under normal conditions, our fault-tolerance support does not introduce extra delay. To show the delay incurred during the connection failover, we isolate those requests that were sent between 25 seconds and 35 seconds into the two experiments. This is 5 seconds before and after the crash. We plot the cumulative distribution of their response times in Figure 9. As we can see, for some requests (log analysis shows 4 requests to be exact), the extra delay incurred by the connection failover is around 2 seconds, which is mostly attributed to the delay caused by the kernel thread. Even in this case, this extra delay is still quite acceptable for most users. Thus, our failover scheme is efficient and virtually transparent to the user.

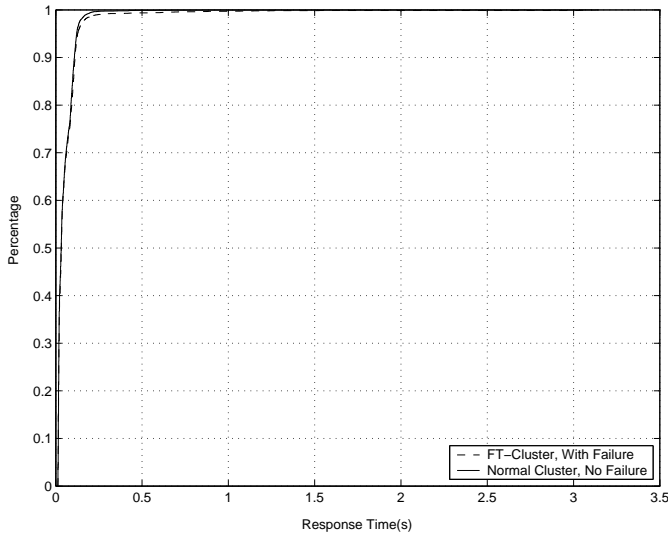


Fig. 8. CDF of the Response Time of All Requests

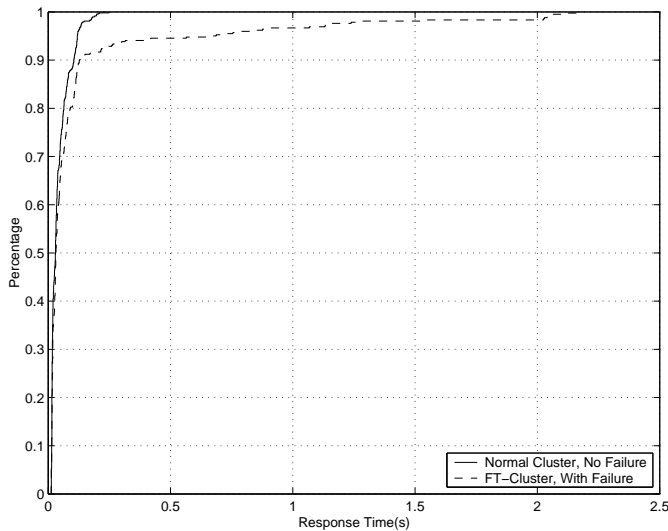


Fig. 9. CDF of the Response Time of the Requests Happened Between 25s and 35s

## V. RELATED WORK

Providing fault tolerance in server clusters has drawn attention from both industry and academia. In [11] a high-availability (clustering) solution is described for Linux. It provides several very useful software packages. One of them is heartbeat, which can monitor the back-ends and inform the front-end when one of them dies. Our heartbeat scheme is similar to it. Another project is [16], which is aiming at health checking for clusters. It implements a framework based on three layer checks: layer3, layer4, and layer5/7. All these can be integrated into our scheme as the failure detection component. Our work is built on [17], which is an open-source project aiming at building a highly scalable and available server cluster. With the help of other software, such as that from [11] and [16], it can achieve certain level of availability:

(1) the load balancer can mask failed back-ends and put them back into service when they come back later, and (2) two load balancers monitoring each other can prevent the load balancer from being a single point of failure. However, when the back-ends fail, the established connections are lost. Our work is complementary to [17].

The work in [2], [3] adopts a standby backup server to achieve client-transparent fault tolerance. Their basic idea is to ensure that the incoming packets are seen by the standby server before the primary server, and a complete copy of the reply is sent to the backup before any packet is sent to the client. Their first implementation [3] is based on a user-level proxy, and thus incurs a high processing overhead. A later version [2] switches to a kernel-level implementation. Its overhead is about 1/6 of the first version. Even in this case, the primary server's throughput is reduced by about 50%. Also, an extra backup server is needed in their scheme. Another solution that needs extra hardware is [10]. At the server, a layer is inserted between the application and TCP, and another layer is inserted between TCP and IP. These two layers, in turn, communicate with a logger, running on a separate machine. The connection states are maintained at the logger. The authors did not mention how to deal with the logger machine's failure. Moreover, a dedicated high speed connection between the server and the logger is critical to keep the overhead on the throughput and latency low. If the dedicated connection between the server and the logger is only 10MB, the throughput can decrease by as much as 30%, and the latency can increase by 5 times. Our scheme outperforms these schemes in terms of lower overhead. Also we don't need dedicated network connections or extra machines.

Another solution to fault-resilience in a web cluster is presented in [4], [5]. The front-end pre-forks a number of persistent connections with the back-ends. When a client tries to connect to the cluster, the front-end first accepts the connection itself, and then binds this connection to one pre-forked connection based on the content the user requests. After that, the front-end acts as a TCP gateway [18]. It rewrites and relays the packets between the user connection and the pre-forked connection. One major drawback of this solution is that every incoming and outgoing packet has to go through the front-end, thus significantly reducing the system throughput since outgoing packets typically outnumber incoming packets. Our approach does not have this problem.

HydraNet-FT [9] proposes an infrastructure for dynamically replicating services across an internetwork and can recover partially processed requests. However, it also has a serious performance issue: when fault tolerance is supported, system throughput can drop by up to 80%.

The authors of [6], [7] introduce Migratory TCP for connection migration. Migratory TCP is not designed to handle connection migration in the case of server failure because it requires voluntary state transfer between the new server and the old server. Moreover, connection migration is initiated by the *client*, and thus is not client transparent. Another example of client-aware connection failover is [8]. The transport layer

at the client side is fully aware of the connection migration, and can even assume the responsibility of picking a new server for the connection. The major limitation of these two schemes is that they require the modification of the TCP stack at the client side, which we think is better to avoid in the common applications we consider.

## VI. DISCUSSION AND FUTURE WORK

In this section, we discuss some additional implementation issues and optimization possibilities. We also describe some directions we plan to explore in the future.

### A. Multiple Backups

Under some situations, it might be useful to have more than one backup per primary server. Our scheme can be easily extended to satisfy this requirement. Since the back-ends back each other up mutually, more than one backup per primary server also means that each back-end has to be backup for more than one primary server. Consequently, each back-end will join multiple multicast groups: one for itself as primary and one as backup for each of its primary servers. A table is needed at each back-end to map a backup connection to its primary server. When a back-end fails, the front-end will first find out all of its backups, notify one of them to take over the connections, and notify the others to discard the connection states maintained for this failed node.

### B. Fail Stop Model

Our current scheme is designed based on the assumption that the server failure follows a fail stop model (i.e., it will not send out any information to disturb the normal operation of TCP connections when it crashes). This is generally true when the server failure is a hardware failure, such as a power failure or a NIC failure. In that case, the server just becomes 'quiet' immediately, which is exactly fail stop. This can also be true when it's caused by some kernel bugs. However, things can be different when it is the application software that fails, e.g., a buffer overflow when parsing an extremely long URL. For most operating systems, when a process terminates, all the files it opened, including the network connections, will be closed. In terms of TCP, a FIN or RST packet will be sent to the client, which essentially closes or aborts the connection. Hence, it is no longer possible to migrate the connection. However, even if we disallow the operating systems from closing the connections in order to migrate them, the effort may not pay off: the same problem can arise again on the new machine and cause another failure once the connection is migrated and the URL parsing resumes. Therefore, connection migration is necessary and beneficial when the failure is caused by hardware.

### C. Optimized Recovery

When the connection is recovered at the backup server, the application generates the reply from the very beginning even if the client has already received part of the reply. It is not a serious problem for most of the Web requests,

since previous studies [19], [20] have shown that most web replies are less than 8K bytes. However, it might introduce considerably longer delays when the requested file is very large.

Two steps are needed to speed up the recovery process for large files<sup>4</sup>:

- 1) When the connection is being recovered, the kernel modifies the HTTP request by adding a RANGE header [22] into it. The range specifies the portion of the file to be sent back to the client. In other words, it excludes the portion of the file that the client has already received. The low end of the range is derived by subtracting the size of the HTTP header within the reply from the number of bytes that the client has acknowledged, and the high end of the range is unspecified, which is allowed by the protocol specification, since the kernel does not know the file size. There are two ways to know the size of the HTTP header within the reply: (1) the kernel scans the data passed by the application, (2) the application notifies the kernel. The first approach is application transparent but is not efficient, since the kernel has to scan every connection and notify the backup server even if a small file is requested. The second one is more flexible and efficient: the application can decide to pass this information to the kernel only when the requested file is very large.
- 2) When the application begins to send out the reply, it first sends out an HTTP reply header. This is not what the client is waiting for, hence, the kernel silently removes this header and passes the remaining data to the client.

We will evaluate these techniques and report the result in the future.

### D. Future Work

Our work assumes that the reply generated again by the backup after the connection is recovered is not different from what was generated originally by the primary. Otherwise, it is hard to 'concatenate' these two replies. A situation can occur where the requested file is modified while connection recovery is in progress. It can also occur when the request is for dynamic content. We plan to investigate this problem in the future. Another direction we plan to explore is to apply this technique to stateful protocols, such as FTP or SSL. These protocols have some application level state information associated with each connection. Simply migrating the connection to the backup server is not enough for these protocols. Apparently, some help from the upper layer protocol or the application may be necessary.

## VII. CONCLUSION

The reliability of the server cluster is critical to the large online service providers. When a back-end server fails, it's better to transparently migrate those connections serviced by

<sup>4</sup>Note that a similar technique can be also applied to fast recovery of FTP connections [21].

it to another working server. In this paper we have proposed a novel scheme to achieve this goal. We have implemented a prototype system, and the evaluation shows that its overhead is very small. Moreover, we do not need any modification to the client side software, and do not need support from any dedicated or extra hardware.

#### ACKNOWLEDGMENT

The work reported in this paper was supported in part by NSF grants CCR-0093144, CCR-0208769, and MURI grant N00014-01-1-0576. The authors would like to thank Chengdu Huang and Zhengdong Yu for their valuable suggestions and help during the experiment. This paper has benefited from the detailed and insightful comments from anonymous reviewers.

#### REFERENCES

- [1] K. Regan, "ebay slammed by 10-hour outage." [Online]. Available: <http://newsfactor.com/perl/story/6461.html>
- [2] N. Aghdaie and Y. Tamir, "Implementation and evaluation of transparent fault-tolerant web service with kernel-level support," in *11th IEEE International Conference on Computer Communications and Networks*, 2002.
- [3] N. Aghdaie and Y. Tamir, "Client-transparent fault-tolerant web service," in *20th IEEE International Performance, Computing, and Communications Conference*, 2001, pp. 209–216.
- [4] C.-S. Yang and M.-Y. Luo, "Realizing fault resilience in web-server cluster," in *International Conference for High Performance Computing and Communications*, 2000.
- [5] M.-Y. Luo and C.-S. Yang, "Constructing zero-loss web services," in *INFOCOM*, 2001.
- [6] K. S. Florin Sultan and L. Iftode, "Transport layer support for highly-available network services," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-429, 2001.
- [7] K. S. Florin Sultan and L. Iftode, "Migratory tcp: Connection migration for service continuity in the internet," in *22nd International Conference on Distributed Computing Systems*, 2002.
- [8] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, "Fine-grained failover using connection migration," in *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.
- [9] G. Shenoy, S. K. Satapati, and R. Bettati, "HYDRANET-FT: Network support for dependable services," in *International Conference on Distributed Computing Systems*, 2000, pp. 699–706.
- [10] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping server-side TCP to mask connection failures," in *INFOCOM*, 2001, pp. 329–337.
- [11] "High-availability linux project." [Online]. Available: <http://www.linux-ha.org>
- [12] "High availability." [Online]. Available: <http://www.linuxvirtualserver.org/HighAvailability.html>
- [13] "Transmission control protocol," RFC 793, Sept. 1981.
- [14] D. Mosberger and T. Jin, "httpperf: A tool for measuring web server performance," in *First Workshop on Internet Server Performance*. ACM, June 1998, pp. 59–67.
- [15] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Measurement and Modeling of Computer Systems*, 1998, pp. 151–160.
- [16] "Healthchecking for lvs & high-availability." [Online]. Available: <http://keepalived.sourceforge.net/>
- [17] "Linux virtual server." [Online]. Available: <http://www.linuxvirtualserver.org>
- [18] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu, "The state of the art in locally distributed web-server systems," *ACM Computing Surveys*, vol. 34, no. 2, pp. 263–311, June 2002.
- [19] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of World Wide Web Client-based Traces," Boston University, CS Dept, Boston, MA 02215, Tech. Rep. BUCS-TR-1995-010, April 1995.
- [20] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM*, 1999, pp. 126–134.
- [21] J. Postel and J. Reynolds, "File transfer protocol(ftp)," RFC 765, Oct. 1985.
- [22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," RFC 2616, June 1999.