Proceedings of the
20th IEEE International Performance, Computing, and Communications Conference,
Phoenix, Arizona, pp. 209-216, April 2001.

Client-Transparent Fault-Tolerant Web Service

Navid Aghdaie and Yuval Tamir
UCLA Computer Science Department
Los Angeles, California 90095
{navid,tamir}@cs.ucla.edu

Abstract

Most of the existing fault tolerance schemes for Web servers detect server failure and route future client requests to backup servers. These techniques typically do not provide transparent handling of requests whose processing was in progress when the failure occurred. Thus, the system may fail to provide the user with confirmation for a requested transaction or clear indication that the transaction was not performed. We describe a client-transparent fault tolerance scheme for Web servers that ensures correct handling of requests in progress at the time of server failure. The scheme is based on a standby backup server and simple proxies. The error handling mechanisms of TCP are used to multicast requests to the primary and backup as well as to reliably deliver replies from a server that may fail while sending the reply. Our scheme does not involve OS kernel changes or use of user-level TCP implementations and requires minimal changes to the Web server software.

1. Introduction

The Internet is often used for transaction based applications such as online banking, stock trading, reservation processing, and shopping, where erroneous processing or outages are unacceptable. This has motivated the development of fault tolerance techniques for increasing the availability and reliability of Internet services. These techniques have included specialized routers and load balancers [2, 3, 10, 11, 20], data replication [5, 24], client-aware server replication [14, 21], and transparent server replication [6, 12, 15, 23].

Many Internet services are provided using a three tier architecture, consisting of: a client web browser, one or more replicated front-end servers, and one or more back-end servers. The front-end server is usually a web server such as the Apache web server, Netscape Enterprise Server, or Microsoft IIS. The web server is responsible for receiving the client requests and replying with a status code, the content of a file, return value from a program/script which it executes, or results from a back-end server with which it communicates. The back-end server is often a database server that does the bulk of the required processing for each client request. The communication between the clients and the front-end servers use the HTTP protocol which is a one-to-one reply/request protocol. The HTTP protocol requires reliable transmission and is therefore typically implemented on top of TCP/IP. The client software — the web browser — is typically developed independently of the web service and it is therefore critical that any fault tolerance scheme used to provide the service be either transparent to the client or rely only on standard features implemented in every web browser in use.

The front-end web servers are typically considered stateless since most of the relevant state is maintained by the back-end servers with the rest maintained by the client, as an HTTP cookie [17] or embedded in a URL. Hence, most fault tolerance schemes for web servers simply provide standby backup servers that can start processing new requests in place of a failed server. The requests that were being processed are effectively dropped. Depending on failure time, the clients may or may not receive a reply back for those requests. To complicate matters, the web server failure can occur before or after the client request reaches the back-end server, leading, respectively, to the loss of the request or its execution by the back-end server. A standard web browser client has no way of determining whether or not the request was processed and it must re-issue the request to be assured of its execution. The re-issuing of the request can also cause other problems since the same request can now be executed multiple times which may lead to undesired results.

The difficulties in dealing with requests in progress when front-end web servers fail are due to the fact that these servers do maintain some state. The most important maintained state is the state of active connections. The web server relays data between the client and the back-end and tracks the location and status of client requests and back-end replies. A stateless backup cannot seamlessly take over open TCP connections and continue with a transaction that was in progress when its front-end server failed. Encryption keys used for secure servers [13] are another important example of front-end server state — a stateless backup will not have the necessary keys for maintaining an existing secure connection. Such front-end server state is often ignored by fault-tolerant web server architectures, but it is critical that this information be preserved by a

backup web server if client-transparent fault tolerance is to be achieved.

We propose a fault tolerance scheme that reduces web service failures by detecting server failures and routing requests to a standby backup server. The scheme is completely transparent to the client. Unlike most other schemes, our approach ensures that all requests, including those being processed at the time of server failure, are processed successfully. The error control mechanisms of TCP are used to provide reliable multicast of client requests to the primary server and the standby backup. These mechanisms are also used to deliver a reply to the client despite server failure. Our implementation is based on simple proxies, does not involve any changes to the OS kernel or using user-level TCP implementations, and requires minimal (if any) changes to the web server software. We describe the design and implementation of our scheme as well as preliminary performance evaluation.

2. Transparent Fault-Tolerant Service

The design and implementation of the system is based on several key assumptions regarding the server hosts and the network connections. As mentioned earlier, our fault tolerance mechanism is based on the standard standby backup scheme [7]. We assume that the primary and backup hosts are fail-stop [22]. Real hosts clearly do not meet this assumption. There is a well-established theoretical foundation for the implementation of fail-stop hosts using hosts with more general (realistic) failure modes [22]. Practical tradeoffs associated with implementing fail-stop hosts are beyond the scope of this paper. Thus, host failure is detected based on periodic heartbeat messages exchanged between the hosts.

We assume that the local area network connecting the two servers as well as the Internet connection between the client and the server LAN will not suffer any *permanent* faults. The primary and backup hosts are connected on the same IP subnet. In practice, the reliability of the network connection to that subnet can be enhanced using multiple routers running protocols such as the Virtual Router Redundancy Protocol [16]. This can prevent the local LAN router from being a critical single point of failure. Our scheme does not currently deal with the possibility that the two hosts become disconnected from each other while both maintaining their connection to the Internet. Forwarding heartbeats through multiple ''third parties'' could be used to detect this situation and continue normal operation in a degraded mode or intentionally terminate one of the servers. We assume that the primary and backup are physically close so that communication between the two servers is much faster than communication with the client. The web server software is running on both the primary and backup. However, during normal operation, requests are processed only on the primary.

The system is client-transparent — clients communicate with a single server address, referred to here as the *advertised address*, and are unaware of server replication. The advertised address consists of an IP address and a TCP port number. At the TCP/IP level, all messages sent by clients have the advertised address as the destination and all messages received by clients have the advertised address as the source address, regardless of the state of the system and the replicated servers. As previously discussed, the primary and backup hosts are connected on the same IP subnet, which is also the subnet of the advertised address.

The basic idea of the scheme is to use the error handling mechanisms of TCP to ensure that the backup has a copy of each request before it is available to the primary. Hence, if the primary fails before transmitting the reply, the backup can process its copy of the request, generate a reply, and send it to the client. Normally, once a reply is generated by the primary, a complete copy is sent to the backup before any reply is sent to the client. After the backup has the reply, the primary sends the reply to the client. If the primary fails *before* starting to transmit the reply to the client, the backup can transmit its copy of the reply to the client. If the primary fails while sending the reply to the client, the error handling mechanisms of TCP are used to ensure that the unsent part of the reply will be sent by the backup.

Each HTTP request and reply consists of one or more TCP packets. The TCP packets can be classified into four categories: client data, client acknowledgment, server data, and server acknowledgment. Data and acknowledgment packets from the same source can be piggy-backed on top of each other for efficiency, but we describe them as separate packets for clarity. As long as both the primary and backup are operational, the advertised address is mapped to the backup. Hence, the backup server receives all packets with the advertised address as their destination. Upon receipt of a packet, the backup server sends a copy of the packet to the primary server by changing the destination address of the packet. The packet's source address remains the client's address. Thus, these packets appear to the primary server as though they were sent directly by the client.

The client expects a TCP acknowledgment packets from the advertised address in response to the data packets that it sends. Although its real address is not the advertised address, the primary server generates the acknowledgment packets to the client, using the advertised address as the source address of these packets. As mentioned earlier, after the processing of the client request and generation of a reply, the primary server sends the entire reply to the backup server. Next, the primary server sends the reply data packets to the client. As with the acknowledgment packets, the reply data packets have the advertised address as their source address. Upon the receipt of the reply data packets, the client sends TCP acknowledgment to the source of the

Table 1: Communication errors and the mechanism used to recover from them. All actions except relays by the backup are a direct result of using TCP.

packet lost/corrupted	detection and recovery mechanism
client data: client → backup	primary doesn't receive packet, no ack to client, client retransmits
client data: backup → primary	primary doesn't receive packet, no ack to client, client retransmits, backup ignores retransmitted packet as duplicate but still relays it to primary
server ack: primary → client	client doesn't receive ack and thus retransmits data packet, primary (and backup) ignore retransmitted data packet as duplicate and primary retransmits ack
server data: primary → client	no ack to primary, primary retransmits server data packet
client ack: client → backup	primary doesn't receive ack, primary retransmits server data packet, client ignores retransmitted packet as duplicate but still retransmits ack
client ack: backup → primary	primary doesn't receive ack, primary retransmits server data packet, client ignores retransmitted packet as duplicate but still sends ack, backup ignores duplicate ack but relays it to primary

Table 2: Recovery from server failures that occur during different phases of handling HTTP requests and replies.

failed server	HTTP message processing phase	recovery mechanism
backup	after backup receives an incomplete client HTTP message	some client TCP data packets of the HTTP message are not acknowledged since they were not received by primary, primary takes over advertised address, client retransmits and/or transmits any unacknowledged TCP packets of the message, primary receives and processes entire message
backup	after backup receives a complete client HTTP message but before all of it is relayed to primary	primary takes over advertised address, since some client TCP data packets are unacknowledged, they are retransmitted by the client, primary receives and processes entire message
backup	after backup forwards a complete client HTTP message to primary	all client TCP data packets are properly acknowledged by primary, primary handles message, primary takes over advertised address and handles future requests
primary	after primary receives an incomplete or complete client HTTP message but before it can acknowledge the last client TCP data packet of the message	unacknowledged client TCP data packets are retransmitted, backup takes over and starts operating in simplex mode, backup acknowledges all unacknowledged client TCP data packets and processes message
primary	after primary receives and acknowledges a complete HTTP request but before it sends a full copy of the HTTP reply to backup	backup takes over and starts operating in simplex mode, backup starts processing its copy of pending requests for which replies have not been transmitted, backup generates and sends HTTP reply to client
primary	after primary sends a complete copy of an HTTP reply to backup but before transmitting some of the reply TCP data packets to client	backup takes over and starts operating in simplex mode, backup starts processing its stored copies of HTTP replies which have not been completely acknowledged by client, backup transmits missing TCP data packets of HTTP reply to client
primary	no HTTP request or reply in progress	backup takes over and starts operating in simplex mode handling all future requests

replies, i.e., the advertised address which is actually mapped to the backup server. The backup server receives the acknowledgments and sends them to the primary server in the same manner as client data packets.

The key to the scheme described above is that the backup server obtains *every* TCP packet (data or acknowledgment) from the client *before* the primary server. Thus, the only way the primary obtains a packet from the client is if the backup already has a copy of the packet. TCP data packets from the primary server are sent to the backup before they are sent to the client. Since all the acknowledgments from the client arrive at the backup before they arrive at the primary, the backup can easily determine which replies or portions of replies it needs to send to the client if the primary fails.

Our system can tolerate single server failure as well as transient communication faults that lead to lost, duplicated, or corrupted packets. Packet loss, corruption, or duplication are handled by TCP's error handling mechanisms. Server failures are detected by heartbeat messages exchanged periodically between the servers. If the backup server fails, the primary server takes over the advertised address and starts operating in simplex mode. If the primary server fails, the backup begins processing

HTTP requests and sending replies in simplex mode. Table 1 summarizes possible communication errors and the mechanisms used to recover from them. Table 2 summarizes the mechanisms used to recover from server failures that occur during different phases of handling HTTP requests and replies.

The communication between the front-end servers and a back-end server, such as a database server, is a mirror image of communication between the client and the front-end servers. Messages sent to and from the back-end are seen by both front-end servers. If the back-end server also uses the fault tolerance scheme described in this paper, no other mechanism is needed. However, since transparency of the fault tolerance scheme is not critical between the front-end and back-end servers, other schemes are possible. For example, the front-end servers may include a transaction ID with each request to the back-end. If a request is retransmitted, it will include the transaction ID and the back-end can use that to avoid performing the transaction multiple times [18].

3. A User-Level Implementation

The scheme described in the previous section requires processing of IP packets in ways that do not match standard kernel-level implementations of TCP. This processing could be accomplished by modifying the kernel-level TCP stack [23]. However, avoiding kernel modifications is advantageous for overall system reliability and for portability. Using the ''raw socket'' interface, it is possible to avoid most kernel processing of packets and perform non-standard modifications of packet headers. However, web server clients use TCP and a key requirement of our work was for the scheme to be client-transparent. Hence, the use of the raw socket interface could lead to a requirement for a full user-level implementation of TCP for communication with the client. Such a user-level TCP implementation could be intergrated with the web server code and modified to provide all the required functionality.

Kernel-level TCP implementations are relatively robust since they are critical to the operation of the system and subject to extensive testing and use by developers and users. On the other hand, user-level TCP implementations are not easily available and often lack the robustness of kernel-level implementations. Our implementation uses proxies to avoid both kernel modifications and user-level TCP implementations while minimizing changes to the servers; all at the cost of additional overhead.

The tasks of the scheme described in the previous section can be divided into two categories: modification of TCP packet headers and operations performed at the HTTP message granularity. In our implementation each of these categories of tasks is implemented by a separate process, henceforth referred to as *proxy*. This leads to a simple modular design and also has the potential to facilitate pipelining (on a multiprocessor) of the handling of requests and replies. The proxy that performs operations at the granularity of IP packets is called the *raw proxy* since it uses the socket interface in the RAW mode to gain access to the packet headers. The proxy that performs operations at the granularity of HTTP messages is called the *TCP proxy* since it uses the socket interface in the STREAM (TCP) mode. The TCP proxy sends and receives complete HTTP messages and buffers HTTP requests and replies for fault recovery.

The system is comprised of a raw proxy and a TCP proxy for the primary server as well as a raw proxy and a TCP proxy for the backup server. We refer to the triple of raw proxy, TCP proxy, and server as a *cluster*. The functionality of the ''primary server,'' as described in Section 2, is implemented by the primary *cluster*. The functionality of the ''backup server,'' as described in Section 2, is implemented by the backup *cluster*. We assume that each of the clusters is either operational or fail stop as a single entity. Using fail-stop hosts [22], this assumption can be enforced using heartbeats from each host running any cluster component coupled with intentional termination of a cluster if any of its components fail. The overall structure of the system is shown in Figure 1.

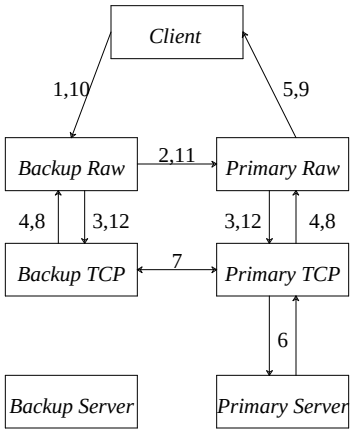

Figure 1: System structure for client-transparent fault tolerance. The connections shown are the TCP/IP packet routes for normal fault-tolerant (duplex mode) operation.

The backup TCP proxy caches HTTP requests from the client (arriving via the backup raw proxy). The primary server gets each request from the primary TCP proxy and sends the HTTP reply back to the primary TCP proxy. From the primary TCP proxy the entire reply is sent to the backup TCP proxy. The backup TCP proxy must then match this reply with the corresponding HTTP client request received previously. Each client HTTP request can be uniquely identified by the client address and a request sequence number. However, since the TCP proxies communicate with the raw proxies and

not directly with the client, the TCP proxy does not automatically have the client address. To solve this problem the raw proxies add the client address to the HTTP requests before forwarding them to the TCP proxies. The primary TCP proxy removes the client address from the HTTP request before it is sent to the primary server. These are items 3 and 6 in the description of TCP level events below (figure 1).

In order to explain the operation of our system, we provide a step-by-step explanation of the progress of requests and replies when the system is operating in its normal fault-tolerant (duplex) mode. The numbers of the steps correspond to the labels in Figure 1.

1. The client sends a data packet to the advertised address, which is mapped to the backup raw proxy.
2. The backup raw proxy reads the packet, changes its destination address to the primary raw proxy (the source address remains the address of the client) and sends the packet (to be received by the primary raw proxy).
3. Each raw proxy changes the source address of the client's packet to its own address. If this is the first data packet from the particular client address, the raw proxy also appends the client IP address and TCP port number to the TCP data and makes appropriate changes to the packet's TCP sequence number. The modified packet is then sent to each raw proxy's respective TCP proxy.
4. The TCP proxies receive the packets sent to them by the raw proxies and the OS kernels on which the TCP proxies are running send TCP acknowledgment packets back to the raw proxies.
5. The raw proxies receive the TCP acknowledgment packets from the TCP proxies. The primary raw proxy changes the source address of the packet to the advertised address and the destination address to the client address. It also modifies the TCP acknowledgment number to account for the extra bytes that are sent to the TCP proxy at step 3 since these extra bytes are acknowledged by the TCP proxies. The primary raw proxy then sends this packet to the client. In duplex mode, the backup raw proxy drops the packet.
6. When enough TCP data packets have arrived at the TCP proxy to compose an entire HTTP request, the TCP proxies remove the extra bytes (client address) that are placed in the request by the raw proxies (step 3). The backup TCP proxy caches the received request. The primary TCP proxy sends the HTTP request to the primary server via a TCP connection. If all goes well, the primary TCP proxy then receives the HTTP reply from the primary server. Note that this entire step can occur simultaneously with step 5.
7. The primary TCP proxy appends the client address that it removed from the HTTP request in step 6 to the HTTP reply and sends the reply to the backup TCP proxy via a TCP connection. The backup TCP proxy receives the HTTP reply and uses the embedded client address to match the reply with the HTTP request that it received in step 3.
8. The TCP proxies send the HTTP reply to their respective raw proxies. The HTTP replies may be broken up (by the OS kernel) into several TCP packets.
9. The raw proxies change the source address to the advertised address, the destination address to the client's address, and adjust the TCP sequence number. Normally, the backup raw proxy discards the packet. The primary raw proxy sends the TCP packets to the client.
   The TCP sequence number adjustment is necessary in order to ensure that both the primary and backup are using the same sequence numbers, allowing a transparent switch to simplex mode operation in case of failure. Each TCP proxy is free to choose any sequence number when initializing a TCP connection. The raw proxies use the same algorithm to generate the actual sequence number that will be used when communicating with the client, and each raw proxy calculates the offset of the actual TCP sequence number and the TCP sequence number generated by the TCP proxy. This offset is then used to modify the TCP sequence number for the lifetime of the connection.
10. The client receives the TCP data packets (HTTP reply) and sends acknowledgment packets to the source address of the data packets, i.e., the advertised address.
11. Similar to step 2, the backup raw proxy receives the acknowledgment packet, changes its destination address to the primary raw proxy (maintaining the client as the source address) and sends the packet.
12. The raw proxies change the TCP acknowledgment numbers to match the TCP sequence numbers used by the TCP proxies (reverse of step 9) and send the packets to their respective TCP proxies.

Since only the primary raw proxy sends packets to the client (step 9), a problem can arise if the backup cluster falls behind the primary and as a result receives TCP acknowledgments for packets that it has not yet sent. To solve this problem the backup raw proxy keeps track of TCP acknowledgment packets that it receives. Whenever it receives a TCP data packet from the backup TCP proxy that contains a TCP sequence number that has already been acknowledged by the client, the backup raw proxy generates a ''fake'' acknowledgment packet with the client address as the source and sends it to the backup TCP proxy. This allows ''old'' data at the backup TCP proxy to be acknowledged.

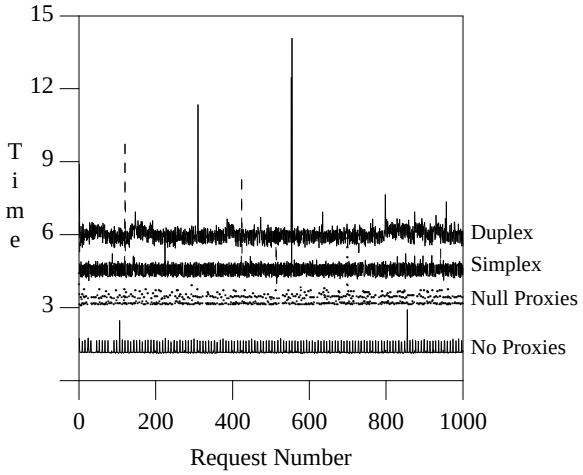

Figure 2: Comparison of client measured request times (in milliseconds) for different system modes

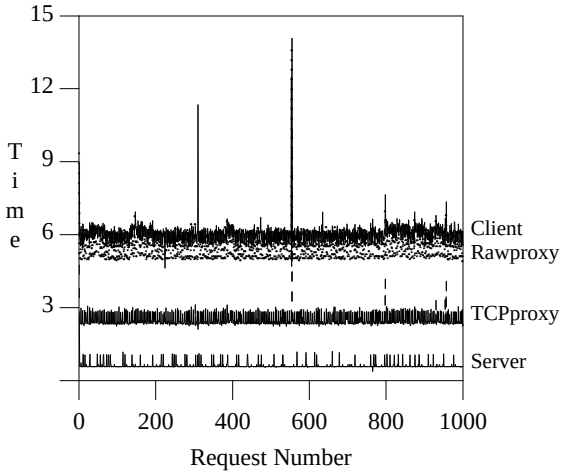

Figure 3: Component breakdown of duplex mode request time (in milliseconds)

Table 3: Average and mean request times for different system modes (figure 2)

Mode	Avg (msec)	Mean (msec)
Duplex Mode	6.05	6.08
Simplex Mode	4.56	4.44
Null Proxies	3.27	3.19
No Proxies	1.23	1.16

Table 4: Average and mean breakdown times for duplex mode (figure 3)

Component	Avg (msec)	Mean (msec)
Client	6.05	6.08
Raw Proxy	5.42	5.26
TCP Proxy	2.52	2.40
Server	0.61	0.57

4. Performance

In this section we show our system's overhead, the breakdown of the overhead within the system, and the effect of message size on our system and non replicated servers. The experiments were performed on 350 MHz dual Intel Pentium II PC's running Solaris 7 and connected via a switched network using 100 Mbit/Sec Ethernet cards and a Nortel 350 Baystack 10/100MB switch. In all experiments the raw and TCP proxies of each cluster (primary/backup) ran on the same machine. We used custom clients and servers similar to those of Wisconsin Proxy Benchmark [1] for our measurements. The client generates continuous HTTP requests and in response the server sends a reply of predetermined size without any processing.

Figure 2 compares the request latency times as measured by the client for 1 kbyte messages. We ran the same experiment with our system in duplex mode and simplex mode. We also ran the experiment with direct client and server connections (''no proxies'') and on a simplex mode system with proxies that just relay the packets (''null proxies''). The performance of the system with null proxies was evaluated in order to evaluate the overhead introduced by our decision to use the two proxy approach. As mentioned earlier, a kernel implementation of our system could greatly reduce this overhead. The difference between the simplex and duplex modes shows the replication overhead. We have verified that the ''spikes'' in figure 2 are not caused by processing time of our code, message loss, or retransmission. We believe these spikes to be caused by elements outside of our implementation, such as the operating system. Table 3 shows the average and mean request times for each experiment.

Figure 3 and table 4 show the breakdown of where the time is being spent for each message. They show the times for the raw proxy, TCP proxy, and the server. Note that the figure depicts an exaggerated view of overhead that is introduced by the proxies since we use a custom server that does not do any processing. In practice, taking into account server processing and Internet communication delays, server response times of hundreds of milliseconds are common. The absolute overhead time introduced by the proxies remains the same regardless of the server response time and therefore our implementation overhead is only a small percentage of the overall response time of practical servers.

The comparison of request times for different HTTP reply message sizes shows that our system scales in the same manner as a non replicated system. Figure 4 shows the client request times for our system in duplex mode and figure 5 shows the times of the same requests for direct client to server communication. The server processing time is also shown in both figures. Table 5 lists the average and mean times. Our system introduces an increased connection setup overhead (difference

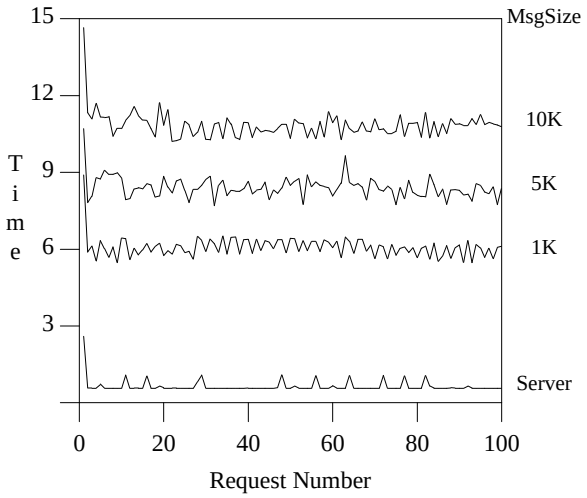

Figure 4: Request times (in milliseconds) for different HTTP message sizes — duplex mode

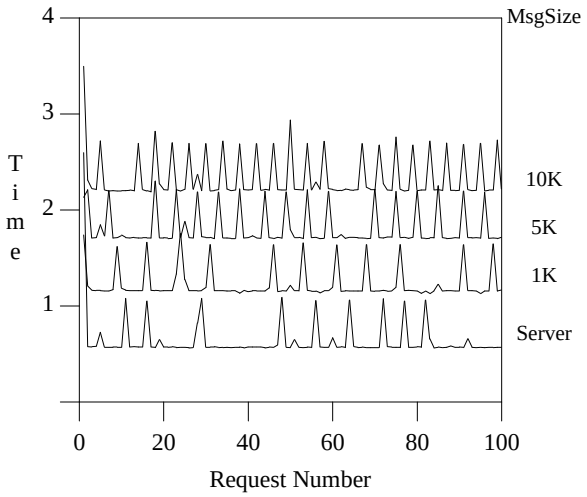

Figure 5: Request times (in milliseconds) for different HTTP message sizes — no proxies

Table 5: Average and mean request times for different HTTP message sizes

Mode	Size (kbytes)	Avg (msec)	Mean (msec)
Duplex	1	6.05	6.08
Duplex	5	8.41	8.36
Duplex	10	10.85	10.87
No Proxies	1	1.23	1.16
No Proxies	5	1.81	1.71
No Proxies	10	2.34	2.21

between server time and small message size) and a slightly larger per packet processing overhead (difference between small and large messages). The figures show that the processing overhead increases linearly in respect to message size in both cases and therefore both systems scale in the same manner with respect to message size and number of TCP/IP packets.

5. Related Work

If server replicas are running on multiple hosts, Round Robin DNS [8] and DNS aliasing methods can be used to provide fault tolerance for Web service by changing the host name to IP address mapping depending on the state of the system. When a server failure is detected, the advertised host name is no longer mapped to the IP address of the failed server host. As a result requests after a failure will not be routed to failed servers. This scheme requires the client to re-issue the request if it does not receive a reply for a request. In practice, client may continue to see the old mapping due to DNS caching at the clients and DNS servers. This reduces the effectiveness of this method since after a failure the client may need to re-issue the request several times before it is routed to a new server. Also in this scheme there is no support for recovering requests that were being processed when the failure occurred.

Centralized schemes, such as the Magic Router [2] and Cisco Local Director [9, 10], require request packets to travel through a central router where they are routed to the desired server. Typically the router detects server failures and does not route packets to servers that have failed. The central router is a single point of failure and a performance bottleneck since all packets must travel through it. Distributed Packet Rewriting [4, 6] avoids having single entry point by allowing the servers to send messages directly to clients and by implementing some of the router logic in the servers so that they can forward the requests to different servers. However this either requires running a scheme such as Round Robin DNS on top or that the clients must know the addresses of all replicated servers. None of these schemes support recovering requests that are in the midst of processing at failure time. Furthermore, these schemes do not allow secure connections to persist following server failure.

Many database vendors such as Oracle [18, 19] offer fault tolerant databases and back-end servers that provide high availability and dependability. These offerings provide a fault tolerant back-end server only and do not handle failures in other parts of the system. Hence, while such back-end servers are necessary components of three-tier Web server architectures, they do not solve the problems addressed in this paper.

There are various server replication implementations that are not client transparent. Most still do not provide recovery of requests that were partially processed. Recent work by Frolund and

Guerraoui [14] does recover such requests. However, the client must retransmit the request to multiple servers upon failure detection and must be aware of the address of all instances of replicated servers. A consensus agreement protocol is also required for the implementation of their ''write-once registers'' which could be costly, although it allows recovery from non fail-stop failures. Our raw proxies can be seen as an alternative implementation of the write-once registers which also provides client transparency.

HydraNet-FT [23] uses a scheme that is similar to ours. It is client-transparent and can recover partially processed requests. The HydraNet-FT scheme was designed to deal with server replicas that are geographically distributed. As a result, it must use specialized routers (''redirectors'') to get packets to their destinations. These redirectors introduces a single point of failure similar to the Magic Router scheme. Our scheme is based on the ability to place all server replicas on the same subnet. As a result, we can use off the shelf routers and multiple routers can be connected to the same subnet and configured to work together to avoid a single point of failure. Since HydraNet-FT uses a hot backup scheme, it can only be used with deterministic servers while our standby backup scheme does not have this limitation. Finally, the HydraNet-FT implementation is based on significant kernel (TCP stack) modifications while our implementation is based on user-level proxies.

6. Conclusion and Future Work

We have proposed a client-transparent fault-tolerant scheme for Web service that can be used with the existing three-tier Internet systems. Our system recovers from server failures by routing future requests to unimpaired servers and also continuing the processing and transmission of requests being processed at failure time. Our implementation is based on simple proxies and avoids kernel modifications or the use of user-level TCP implementations. Preliminary performance evaluation indicates that the added latency overhead is insignificant compared to typical Internet delays.

Future work will involve using an actual Web server. Small modification to the server will be requires in order to support persistent secure connections. The performance overhead in terms of lost computation cycles and reduced throughput will also be evaluated.

References

1. J. Almeida and P. Cao, ''Wisconsin Proxy Benchmark,'' *Technical Report 1373, Computer Sciences Dept, Univ. of Wisconsin-Madison* (April 1998).
2. E. Anderson et al., ''The Magicrouter, an Application of Fast Packet Interposing,'' *Class Report - http://www.cs.berkeley.edu/~eanders/projects/magicrouter/* (May 1996).
3. D. Andresen et al., ''SWEB: Towards a Scalable World Wide Web Server on Multicomputers,'' *Proccedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, pp. 850-856 (April 1996).
4. L. Aversa and A. Bestavros, ''Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting,'' *Proceedings of the 2000 IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, pp. 24-29 (February 2000).
5. S. M. Baker and B. Moon, ''Distributed Cooperative Web Servers,'' *The Eighth International World Wide Web Conference*, Toronto, Canada, pp. 1215-1229 (May 1999).
6. A. Bestavros et al., ''Distributed Packet Rewriting and its Application to Scalable Server Architectures,'' *Proceedings of the International Conference on Network Protocols*, Austin, Texas, pp. 290-297 (October 1998).
7. A. Borg et al., ''A Message System Supporting Fault Tolerance,'' *9th Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).
8. T. Brisco, ''DNS Support for Load Balancing,'' *IETF RFC 1794* (April 1995).
9. Cisco Systems Inc, ''Failover Configuration for LocalDirector,'' *Cisco Systems White Paper - http://www.ieng.com /warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm*.
10. Cisco Systems Inc, ''Scaling the Internet Web Servers,'' *Cisco Systems White Paper - http://www.ieng.com /warp/public/cc/pd/cxsr/400/tech/scale_wp.htm*.
11. D. M. Dias et al., ''A scalable and highly available web server,'' *Proceedings of IEEE COMPCON '96*, San Jose, California, pp. 85-92 (1996).
12. C. Fetzer and S. Mishra, ''Transparent TCP/IP based Replication,'' *The 29th International Symposium on Fault-Tolerant Computing - Fast Abstracts*, Madison, Wisconsin (June 1999).
13. A. O. Freier et al., ''The SSL Protocol Version 3.0,'' *http://www.netscape.com/eng/ssl3/ssl-toc.html* (March 1996).
14. S. Frolund and R. Guerraoui, ''Implementing e-Transactions with Asynchronous Replication,'' *IEEE International Conference on Dependable Systems and Networks*, New York, New York, pp. 449-458 (June 2000).
15. C. T. Karamanolis and J. N. Magee, ''Configurable Highly Available Distributed Services,'' *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, Bad Neuenhar, Germany, pp. 118-127 (September 1995).
16. S. Knight et al., ''Virtual Router Redundancy Protocol,'' RFC 2338, IETF (April 1998).
17. D. Kristol and L. Montulli, ''HTTP State Management Mechanism,'' *IETF RFC 2109* (February 1997).
18. Oracle Inc, *Oracle8i Distributed Database Systems - Release 8.1.5*, Oracle Documentation Library (1999).
19. Oracle Inc, ''Oracle Parallel Server: Solutions for Mission Critical Computing,'' *An Oracle Technical White Paper - http://www.oracle.com/database/documents/ parallel_server_twp.pdf* (February 1999).
20. RND Networks, ''Web Server Director for Distributed Sites (WSD-DS),'' *RND Networks Technical Applicaton Note 1035 - http://www.rndnetworks.com/archive/pdfs/whitepapers/ app1035.pdf*.
21. M. Sayal et al., ''Selection Algorithms for Replicated Web Servers,'' *Performance Evaluation Review - Workshop on Internet Server Performance*, Madison, Wisconsin, pp. 44-50 (June 1998).
22. F. B. Schneider, ''Byzantine Generals in Action: Implementing Fail-Stop Processors,'' *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).
23. G. Shenoy et al., ''HydraNet-FT: Network Support for Dependable Services,'' *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 699-706 (April 2000).
24. R. Vingralek et al., ''Web++: A System For Fast and Reliable Web Service,'' *Proceedings of the USENIX Annual Technical Conference*, Sydney, Australia, pp. 171-184 (June 1999).