Proceedings of the International Conference on Parallel and Distributed Computing and Systems
Marina del Rey, California, November 2003.

FAST TRANSPARENT FAILOVER FOR RELIABLE WEB SERVICE

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

ABSTRACT

Fault tolerance schemes can be used to increase the availability and reliability of network services. One aspect of such schemes is *service failover* — the reconfiguration of available resources and restoration of state required to continue providing the service despite the loss of some of the resources and corruption of parts of the state. We have previously presented CoRAL, a fault tolerance scheme for Web service based on a redundant standby backup server and logging. The focus of this paper is the implementation and evaluation of client-transparent failover for this scheme. In the event of a primary server failure, active client connections failover to a spare where their processing continues seamlessly. If extra server resources are available, a new server can be reintegrated into the system to reestablish fault-tolerant operation. Our performance results indicate short failover times and low overhead during fault-free operation.

KEY WORDS
Fault-Tolerant Systems, Network Services, TCP.

1. Introduction

The failure of a server or of the network may cause a network service to fail to respond to ongoing requests and to no longer be available to new requests. For critical network services, fault tolerance techniques can be used to detect failures, identify faulty components, and reconfigure systems and networks to resume operation without the failed components. A key step in any such fault tolerance scheme is the resumption of normal operation with a repaired system state. This step is often called *service failover*.

Web clients and servers are typically developed independently and there is a vast installed base of clients. Hence, client transparency is a critical property of techniques for enhancing the reliability and availability of Web service. We have previously presented CoRAL, a fault-tolerant Web service scheme based on **Co**nnection **R**eplication and **A**pplication-level **L**ogging [1, 2]. CoRAL recovers in-progress requests and does not require deterministic servers or changes to the clients. The focus of our previous work has been on maintaining the state information required to allow service failover and on evaluating the overhead of our scheme during normal operation.

The focus of this paper is on the implementation and evaluation of the failover procedure that is part of our fault tolerance scheme. When an active server fails, the failure is detected and client-server TCP connections transparently failover to a backup server. Requests in midst of transmission at failure time are received and processed by the backup. Those in midst of processing on the primary are reprocessed by the backup. The transmission of partially transmitted replies to the client is completed by the backup. After the initial failover of existing connections to the backup, our scheme is capable of integrating a new server to transparently restore the system to its original primary/backup configuration for new connections. This maintains the fault tolerance characteristics of the system after failover.

The key requirements and major design choices for preserving service state despite server failure are discussed in Section 2. The implementation details of our scheme are explained in Section 3. Performance evaluation results are presented in Section 4. Section 5 discusses related work.

2. Preserving Service State

Fault tolerance schemes for network services may increase the *availability* of the service by providing the ability to service *new* requests following server failure. Such schemes may also increase the *reliability* of the service by ensuring that even requests that are in progress at the time of server failure will be handled correctly. The complexity and overhead associated with providing the increased availability and reliability are dependent on whether the service is *stateless* or *stateful*. For a stateful service, proper operation requires the fault tolerance mechanism to preserve the state despite server failure.

The simplest fault tolerance mechanism provide increased availability for stateless service. In this case, there is no state to be preserved or repaired and there is no attempt to handle requests that are in progress at the time of server failure. Some mechanism for detecting server failure, such as heartbeats, is needed. Failover is accomplished by mapping (routing) new requests to alternate resources (servers). New servers are thus allowed to take over the *identity* of the failed server. In the Web service context, this is typically accomplished by using DNS to change the binding of server name to IP address at the client [6] or by routing the packets with a fixed destination IP address to different servers at the server site [5, 11].

For stateful services, failover requires a valid, up-to-date state to be established at any alternate servers used following server failure. There are three common approaches for preserving the server state: active replication, message logging, and checkpointing. Each approach requires slightly different steps to be taken for failover.

Active Replication — client requests are simultaneously processed by multiple replicas [7, 18]. In normal operation, the output generated from only one of the replica servers, the primary, is sent to the client, while the other outputs are discarded. Should a fault occur, a backup replica takes over for the primary and sends its output to the client. Hence, transparent failover for active replication schemes involves the taking over of the identity of the failed server and reconfiguring a backup replica to communicate with the clients, i.e., transmit its output to the client instead of discarding it.

Message Logging — all incoming client messages are redundantly stored (logged). If a server fails, logged messages are replayed to an alternate server, bringing the alternate server back to the pre-fault state of the failed server [4]. Depending on the goals and requirements of a scheme, the logging may be implemented at different levels. For example, an implementation may log messages at the network transport (TCP) level [4] or at the application level [1, 2]. To achieve transparent failover, all logged messages must be replayed on an initialized server configured with the same identity as the failed server.

Checkpointing — an up-to-date copy of the server state is maintained on a standby backup or in stable storage. For transparent failover, the checkpointed state must be recovered to a replica server with the same identity as the failed server.

The service state potentially consists of two key components: the state of the connection and the state of the service application. In order to properly handle in-progress requests despite server failure, failover requires correct handling of requests that have been partially received prior to failure or replies that have been partially transmitted to the client prior to server failures. The partially received requests and partially transmitted replies are part of the connection state at the server. For the fault tolerance scheme to be client-transparent, this connection state must be available or derivable at an alternate server if the original server fails without the help of the client. If the proper handling of one request depends on previous requests, the service *application* has state. If this is the case, failover requires that an up-to-date version of the service application state be available to the new server following failover.

Web service is typically provided using HTTP over TCP. The TCP connection has state that must be preserved. For example, once bytes sent by the client are acknowledged, they must not be lost. In most cases, Internet services are provided using a three tier architecture, consisting of: a client Web browser, a front-end server (e.g., the Apache Web server), and a back-end server (e.g., a database server). Based on the definition above, the Web service application typically does have state. However, most of that state is stored in the back-end servers. The Web server (the front-end server) state is limited, consisting of the complete (HTTP) requests received for which replies have not been sent.

Our CoRAL scheme [1, 2] preserves service state using a combination of active replication and message logging. The state of the TCP connection is preserved using active replication — the TCP stacks on both the primary and backup process every packet. Message logging at the HTTP level is used to establish the correct Web service state on a backup server following primary server failure. Thus, failover with our scheme requires identity takeover, reconfiguration of a backup replica to become active, as well as possible replay of logged messages at the application level.

3. CoRAL Failover

We have implemented CoRAL using a Linux kernel module and an Apache Web server module on each server replica [2]. The kernel module performs operations at the TCP packet granularity, actively replicating the connection state. It implements reliable multicast of client requests to both replicas and provides a transparent mechanism for the continued transmission of replies in the event of a failure. In fault-free operation, the backup replica forwards copies of client packets to the primary replica. All incoming and outgoing packets are processed at the TCP level on both replicas. The user-level server module performs operations at application level message (HTTP) granularity. The implementation consists of enhancements to the Apache request handling routines and addition of user-level processes to provide application-level logging functionality. In fault-free operation, only the primary server executes the server application, processing the client requests and generating replies. The backup server logs the client request and server reply messages. Requests are kept until the corresponding reply is received from the primary. Replies are kept until the client has acknowledged their receipt at the TCP level.

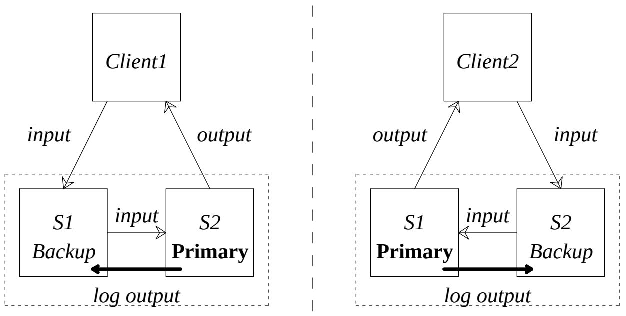


Figure 1: Dual-role operation. Each server replica can simultaneously function as a primary for some clients and backup for others.

Since the server application is only executed on the primary, the primary is likely to require more processing than the logging that is performed on the backup. This can result in unbalanced utilization of the primary and server nodes, especially for processor-intensive applications, such as serving dynamic content. Our dual-role implementation [3] allows each node to serve as primary for some requests and backup for others (Figure 1), ensuring efficient utilization of available processing resources.

We assume failures to be fail-stop [17]. For fault detection, heartbeat messages are exchanged between servers. A process in the user-level server modules of each server periodically sends sequenced UDP packets to its counterpart. Consecutive missed heartbeats signal that a fault has been detected. Hence, the heartbeat rate directly relates to the fault detection time.

In the rest of this section we present the details of service failover by a dual-role duplex system, and restoration of fault-tolerant operation after a failure.

3.1. Service Failover

When a fault is detected, the system must transition from standard replicated (duplex) operation to single server (simplex) mode. The identity of the service is preserved for new and existing connections. Existing active connections are migrated to simplex mode. For each received HTTP request, the system ensures that the complete HTTP reply message is delivered to the client.

In dual-role configuration, each server replica is configured to receive client packets and act as backup for some of the requests [3]. The service address used by the clients that send these packets must continue to be available following a server failure. Hence, the remaining member of the duplex pair has to take over that IP address. This takeover can be implemented using a Linux *ioctl* that establishes an additional IP address alias for the network interface. However, as discussed later in this section, this functionality is included in a new system call that we implemented.

In a standard modern networking setup, the servers are connected to a switched LAN and all packets from remote clients pass through multiple routers on their way to the server. Following IP address takeover, the local router must be informed that a new host (MAC address) should now receive packets sent to the ‘‘migrated’’ IP address. We use gratuitous ARP [16] to accomplish this task. Specifically, ARP reply packets are sent to hosts (including the router) on the same IP subnet without waiting for explicit ARP requests. Once the router receives an ARP reply packet, its ARP cache is updated, causing it to route packets that are sent to the service address to the surviving server. Gratuitous ARP is not reliable since the ARP reply packets may be lost. Hence, we added a level of reliability by pinging a known host outside the server subnet. If a ping reply is received, it implies that the router’s ARP cache has been updated. If a reply is not received within a timeout interval, the gratuitous ARP reply packets are retransmitted. Only a single outside host is used in our implementation. However, this approach can be trivially extended to ping multiple outside hosts to avoid the possibility of the outside host becoming a single point of failure. A ping reply from any outside host would indicate a successful router ARP cache update.

Our implementation uses modules at both the kernel and user levels [2]. A failed server replica is detected by a user-level process running on the other replica and monitoring heartbeats. The kernel module must be informed in order to transition active connections from duplex to simplex processing mode. Hence, we have implemented a system call which allows a user-level process to notify the kernel of a fault detection. The kernel module’s transition from duplex to simplex mode is simple. Before the transition, the surviving dual-role node may be handling connections as both ‘‘primary’’ and ‘‘backup’’. For ‘‘backup’’ connections, the kernel module no longer forwards the incoming client packets to a primary. Also, outgoing packets are sent to the client instead of being discarded. As a result, unacknowledged portions of any logged replies will reach the clients. For ‘‘primary’’ connections, incoming packets are received directly from the clients instead of being forwarded from the backup, but this change is not noticeable to the server.

At the user-level, our server module implementation is composed of multiple processes [2]. The heartbeat process must notify all other user-level processes of a detected fault. The user-level notification is implemented by setting a global flag in shared memory. Each server module process checks this flag when it receives a client request, and determines whether the system is in duplex or simplex mode. At failure time, some of the server module processes may be waiting for an event. For example, a backup process may be waiting for a reply, or a primary process may be waiting for acknowledgement that its reply has been logged. In normal operation, these ‘‘worker processes’’ wait for messages that are eventually delivered to them by a ‘‘demux process’’ that communicates with them via data structures and semaphores in shared memory [2]. When a fault occurs, these waiting processes must be notified to no longer wait, since the events they are waiting for will not occur in post-fault (simplex) operation. The process that detects the fault performs these notifications using the same mechanism normally used by the ‘‘demux process.’’

After a failover and transition to simplex mode, new requests are processed in simplex mode and replies are sent directly to the client. In-progress requests logged at the Backup, for which a reply was never logged (by the failed Primary), are similarly processed in simplex mode.

3.2. Restoration of Fault-Tolerant Service

After a failover, it is desirable to restore the system back to its replicated configuration so that other faults can be tolerated. Our implementation allows for the integration of a new server into the system without any disruption to active connections. No extra resources other than a process listening for a connection are used while operating in simplex mode. After a failover to simplex mode, a server module process listens for a new server that may want to join the system. The new server trying to join the system is initialized with all the required modules and processes for duplex operation before contacting the active simplex server. The new server connects to the waiting simplex server process and the two exchange identity and configuration information. At this point, the active simplex server begins the transition to duplex mode. At the user-level, processes required for the logging of replies in duplex mode are spawned and initialized. Their initialization includes setting up the required TCP connections with the new server. Once all processes are initialized, the active simplex server invokes our system call, notifying the kernel module to also transition to duplex mode. Packets for new connections received after the transition are processed in duplex mode.

Our implementation does not transition the existing client connections being processed in simplex mode to duplex mode. Hence, a server may need to operate in simplex and duplex modes simultaneously. To support multiple modes simultaneously, the kernel module keeps track of the system mode at a per connection granularity. At user-level, the server module uses our system call to check the mode of a connection. This check, i.e. system call, is only necessary while the transition from simplex to duplex is not fully complete — i.e., there are still existing simplex connections that have not yet terminated.

A conceivable improvement to our scheme would be to transition existing simplex connections to duplex mode as well, thus allowing for tolerance of multiple faults on the same connection. The simplex node's existing connection state and generated HTTP replies would have to be transferred to the new node, which can be expensive. We extended our implementation to minimize the need for such a scheme by using the protocol semantics of HTTP/1.1 to ensure that after transition to duplex mode, existing simplex connections will be short-lived. HTTP/1.1 allows multiple requests to be pipelined on the same connection. However, the server has the option of terminating the connection at any point [9]. The server may close the connection cleanly using the HTTP ‘‘Connection: close’’ directive or by closing the connection at the transport level. The client should retry the pipelined requests for which it does not receive a reply [9]. We use this property to our advantage and force leftover simplex connections to be short-lived. After a server integration, the first request on each existing simplex connection is processed normally in simplex mode. However, when sending the reply, we notify the client at the HTTP level that the connection should be closed. As a result, the client will start a new connection for the processing of the rest of pipelined requests. Since these will be new connections, they will be automatically processed in duplex mode by our system. Hence, after the start of transition to duplex mode, our implementation will only process the single current active HTTP request of a connection in simplex mode. The rest of the requests will be processed in duplex mode, on a new connection.

4. Performance Evaluation

Our measurements were performed on 350 MHz Intel Pentium II PCs interconnected by a 100 Mb/sec switched network. The servers were running our modified Linux 2.4.2 kernel and the Apache 1.3.23 web server with logging turned on. In normal operation, servers used the dual-role configuration where each server can simultaneously act as both primary and backup for different requests [3]. Faults were injected in to the system by physically disconnecting a server host from the network, effectively emulating a failstop fault. The server replies were generated dynamically using WebStone 2.0 benchmark's CGI workload generator [15, 20]. This benchmark produces dynamic content by randomly generating each reply byte. The workload is generated by eight client processes, each continuously sending an HTTP request for a 1 Kbyte reply and waiting for the reply before sending the next request.

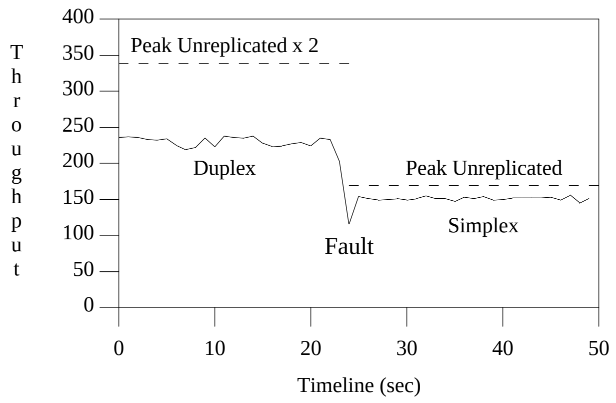


Figure 2: System throughput (in requests per second) for 1 kbyte HTTP replies before and after a fault. Dashed lines indicate the upper bound, i.e., the peak throughputs of one and two unreplicated servers serving the same content.

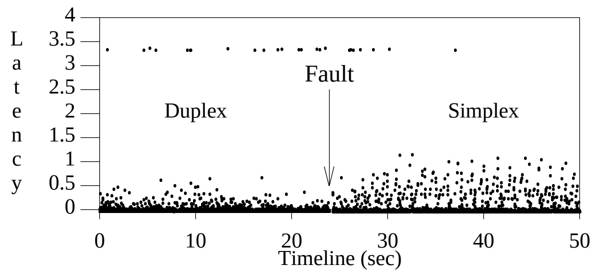
4.1. Service Throughput

Figure 2 shows the system throughput before and after a failover, measured at one second intervals. The system operates at the maximum duplex system throughput prior to the fault. Since in duplex mode replies are not actually generated on the Backup, two servers in duplex mode can process more requests per unit time than a single server in simplex mode. Thus, the throughput of the system decreases following reconfiguration to simplex mode.

4.2. Failover Latency

Figure 3 shows typical client observed request latencies before and after a fault. Although clients continuously transmit requests, there is a gap at the fault point where no requests are processed — the system is unavailable for a short period of time. This is the time required for fault detection and failover from duplex to simplex mode. Our measured failover transition time from fault detection in duplex mode to system execution in simplex mode is just 1.5 milliseconds (see Table 1). Since the transition time is short, most of the overall failover time is the time for fault detection. The heartbeat rate determines the fault detection time but generating and processing heartbeats uses CPU resources. Hence, the choice of this rate is a tradeoff between failover speed and overhead during normal operation. For our evaluation, we used a heartbeat rate of 100 milliseconds.

Unfortunately few implementations have published failover times for comparison. FT-TCP [4] reports log replay times of 485 msec to 2.5 seconds per 1 MB of logged data, depending on network speed. The application re-execution time, to recover application state



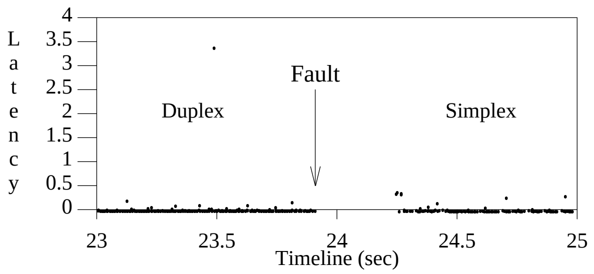


Figure 3: Client observed request latencies (in seconds) for 1 kbyte HTTP replies before and after a fault. Figure on the right is focused at the fault point. During failover, the system is unavailable for a fraction of a second.

Operation	Latency (msec)
Kernel Notification and Operations	0.3
User-level Process Notification and Operations	0.5
Identity Preservation	
gratuitous ARP	0.5
outside host ping	0.2
Total	1.5

Table 1: Breakdown of failover latency measured from the time a fault is detected to start of system operation in simplex mode. This table does not include the time to regenerate replies that were not logged prior to the failure.

and produce replies, is not included and would increase these times further. Zero-Loss Web Services [14] shows system throughput stabilizing and recovering to pre-failure levels within 7 seconds of a fault. Active replication schemes [7, 18] should achieve failover times similar to or better than ours, although we could not find published results.

4.3. Server Integration After Failover

Table 2 shows a breakdown of latency for integration of a new server with a simplex server. Most of the overall time is due to forking of the processes that are used for communication between the server for reply logging [2]. The fork and initialization overhead can be reduced by eliminating the need to fork processes during server integration. This can be accomplished, at a cost of increased resource usage, by forking these processes in simplex mode before they are actually needed.

The other significant amount of time is due to exchange of configuration information, i.e., server TCP/IP addresses, over a TCP connection. This is necessary because we assumed that the active simplex server has no knowledge regarding the new server that is joining it. Static configuration of server pairs can reduce or eliminate this overhead. The identity restoration step consists of removing an aliased IP address from the simplex node, mapping this IP address to the new server, and informing the router using our reliable gratuitous ARP implementation described in section 3.

Operation	Latency (msec)
Identity Restoration	0.3
Exchange Configuration Info Over TCP	3.9
Fork and Initialize Processes	6.8
Total	11.0

Table 2: Breakdown of latency for integration of a new server with a working simplex server node to recover back to a duplex system.

5. Related Work

Many reliability solutions are not transparent and do involve the client in replication and failover. Client involvement varies from simply retrying requests after a timeout period [8, 10] to being directly involved in server selection and migration of active connections [19]. Although client-aware solutions provide flexibility in failover implementation, unless standards such as WS-Reliability [8] are widely accepted, client-transparent solutions are imperative.

Failover for soft-state network elements such as routers typically only consists of migrating the failed element's identity (i.e. IP address) to a replica [12, 13]. Unlike routers, most servers do maintain some connection and/or application state that must be preserved for increased reliability. Existing availability solutions however, usually ignore the need to preserve the server state. Load balancers and centralized director schemes [5, 6, 11] typically monitor the health of a pool of replicas and distribute client requests among healthy servers. If a server fault is detected, new client connections are no longer directed to that server. With this approach, the probability of a new connection being serviced is increased, however active connections at failure time are effectively lost. Hence, these solutions increase the availability of the server, but not its reliability.

FT-TCP [4] is a solution which uses the log and replay approach. All incoming client packets and locally generated TCP connection state are logged to another node. After a fault, logged packets are replayed and reprocessed on a new server with the same identity. It is assumed that the server application is deterministic, so that the replaying of the packets on a new re-initialized replica will lead to the regeneration of exact copies of application output. An extended implementation of FT-TCP [21] also logs application system call results, and is capable of working with non-deterministic applications. Log and replay solutions suffer from relatively long failover times since the log compiled during the lifetime of a connection must be replayed. The failover time can be improved at a cost of added processing and latency during normal operation by preserving the application state. Periodic checkpointing of application state can reduce the failover time since only the portion of logs compiled after the last checkpoint need to be replayed.

Active replication schemes [7, 18] maintain active replicas that execute identical applications using identical inputs. In normal operation, the output of the backup

Scheme	Design Choice	Reported Overhead	Notes
CoRAL [2]	Active Replication + Logging	30%	TCP level replication, HTTP level logging
FT-TCP [4]	Logging	28% - 76%	Logger node resources not included in overhead Application level overhead not considered 1% - 8% overhead if server-logger link is high speed
HotSwap [7]	Active Replication	54.3%	Presented as 9.6% overhead over single server
HydraNet-FT [18]	Active Replication	50% - 90%	Shows throughput reaching 20 - 100% of single server

Table 3: Throughput overhead of several transparent failover schemes. The ''Reported Overhead'' column was derived from published results and shows the percent reduction from the throughput of two standard servers. Due to differences in assumptions, server application functionality, and measurement methodology, direct quantitative comparison of these reported overheads is not meaningful.

server is simply discarded. For failover, the backup simply takes over the identity of the failed server and transmits its output to the clients instead of discarding it. As a result, the failover time for active replication schemes are inherently short. The comparison of failover times for the logging and primary-backup (active replication) versions of FT-TCP [21] validate this point.

For active replication schemes there is at least a 50% processing overhead since a backup node must, at a minimum, perform all tasks executed by the primary (Table 3). The logging approach typically incurs smaller throughput overhead, especially if a dedicated or high speed link between replicas is available. CoRAL uses a combination of the two approaches: active replication at connection (TCP) level, and message logging at application (HTTP) level. For processor-intensive applications, it incurs throughput overheads similar to logging schemes, while achieving short failover times similar to active replication schemes.

6. Conclusion

We have presented the implementation and evaluation of transparent failover for our CoRAL reliable web service scheme. During fault-free operation, our scheme actively replicates the connection state and uses message logging to preserve the application output and support non-deterministic stateless applications. We described practical aspects of the implementation at the kernel, user, and network levels. The analysis of our system's failover latency shows that most required operations are at least an order of magnitude faster than typical Internet latencies. The most significant portion of the failover latency is due to fault detection. By making the fault detection fast compared to typical Internet latencies, the failover time can become invisible to clients. We also presented a mechanism for restoring fault-tolerant operation after a failover. If extra server resources are available, a new server can be reintegrated back into the system, thus allowing the system to tolerate more faults. Our results indicate that our system's window of vulnerability, i.e. the time from fault-detection to restoration of fault-tolerant service, is on the order of few milliseconds beyond the fault detection time.

References

[1] N. Aghdaie and Y. Tamir, ''Client-Transparent Fault-Tolerant Web Service,'' *20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, AZ, pp. 209-216 (April 2001).

[2] N. Aghdaie and Y. Tamir, ''Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support,'' *11th IEEE International Conference on Computer Communications and Networks*, Miami, FL, pp. 63-68 (October 2002).

[3] N. Aghdaie and Y. Tamir, ''Performance Optimizations for Transparent Fault-Tolerant Web Service,'' *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada (August 2003).

[4] L. Alvisi et al., ''Wrapping Server-Side TCP to Mask Connection Failures,'' *IEEE INFOCOM*, Anchorage, Alaska, pp. 329-337 (April 2001).

[5] L. Aversa and A. Bestavros, ''Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting,'' *IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, pp. 24-29 (February 2000).

[6] T. Brisco, ''DNS Support for Load Balancing,'' RFC 1794, IETF (April 1995).

[7] N. Burton-Krahn, ''HotSwap - Transparent Server Failover for Linux,'' *USENIX LISA '02: Sixteenth Systems Administration Conference*, Berkeley, California, pp. 205-212 (November 2002).

[8] C. Evans et al., ''Web Services Reliability (WS-Reliability),'' http://www.oasis-open.org/committees/wsrm/charter.php, OASIS Web Services Reliable Messaging Technical Committee, Working Draft (January 2003).

[9] R. Fielding et al., ''Hypertext Transfer Protocol -- HTTP/1.1,'' RFC 2616, IETF (June 1999).

[10] 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).

[11] Intel Inc, ''Intel NetStructure e-Commerce Products,'' *http://www.intel.com/support/netstructure/commerce/index.htm*.

[12] S. Knight et al., ''Virtual Router Redundancy Protocol,'' RFC 2338, IETF (April 1998).

[13] T. Li et al., ''Cisco Hot Standby Router Protocol (HSRP),'' RFC 2281, IETF (March 1998).

[14] M.-Y. Luo and C.-S. Yang, ''Constructing Zero-Loss Web Services,'' *IEEE INFOCOM*, Anchorage, Alaska, pp. 1781-1790 (April 2001).

[15] Mindcraft Inc, ''WebStone Benchmark Information,'' *http://www.mindcraft.com/webstone*.

[16] D. C. Plummer, ''An Ethernet Address Resolution Protocol,'' RFC 826, IETF (November 1982).

[17] F. B. Schneider, ''Byzantine Generals in Action: Implementing Fail-Stop Processors,'' *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).

[18] G. Shenoy et al., ''HydraNet-FT: Network Support for Dependable Services,'' *20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 699-706 (April 2000).

[19] A. C. Snoeren et al., ''Fine-Grained Failover Using Connection Migration,'' *3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, pp. 221-232 (March 2001).

[20] G. Trent and M. Sake, ''WebSTONE: The First Generation in HTTP Server Benchmarking,'' *http://www.sgi.com /Products/WebFORCE/WebStone/paper.html* (February 1995).

[21] D. Zagorodnov et al., ''Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP,'' *IEEE Dependable Computing and Communications Symposium (DSN-2003)*, San Francisco, California, pp. 393-402 (June 2003).