

Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support

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

Abstract—Most of the techniques used for increasing the availability of web services do not provide fault tolerance for requests being processed at the time of server failure. Other schemes require deterministic servers or changes to the web client. These limitations are unacceptable for many current and future applications of the Web. We have developed an efficient implementation of a client-transparent mechanism for providing fault-tolerant web service that does not have the limitations mentioned above. The scheme is based on a hot standby backup server that maintains logs of requests and replies. The implementation includes modifications to the Linux kernel and to the Apache web server, using their respective module mechanisms. We describe the implementation and present an evaluation of the impact of the backup scheme in terms of throughput, latency, and CPU processing cycles overhead.

I. INTRODUCTION

Web servers are increasingly used for critical applications where outages or erroneous operation are unacceptable. In most cases critical services are provided using a three tier architecture, consisting of: client web browsers, one or more replicated front-end servers (e.g. Apache), and one or more back-end servers (e.g. a database). HTTP over TCP/IP is the predominant protocol used for communication between clients and the web server. The front-end web server is the mediator between the clients and the back-end server.

Fault tolerance techniques are often used to increase the reliability and availability of Internet services. Web servers are often stateless — they do not maintain state information from one client request to the next. Hence, most existing web server fault tolerance schemes simply detect failures and route *future* requests to backup servers. Examples of such fault tolerance techniques include the use of specialized routers and load balancers [4, 5, 12, 14] and data replication [6, 28]. These methods are unable to recover in-progress requests since, while the web server is stateless *between* transactions, it does maintain important state from the arrival of the first packet of a request to the transmission of the last packet of the reply. With the schemes mentioned above, the client never receives complete replies to the in-progress requests and has no way to determine whether or not a requested operation has been performed [1, 15, 16] (see Figure 1).

Some recent work does address the need for handling in-progress transactions. Client-aware solutions such as [16, 23, 26] require modifications to the clients to achieve their goals. Since many versions of the client software, the

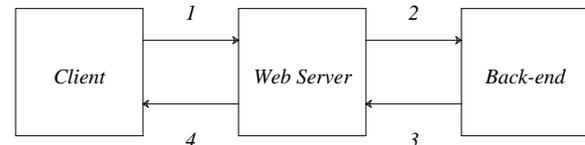


Figure 1: If the web server fails before sending the client reply (step 4), the client can not determine whether the failure was before or after the web server communication with the back-end (steps 2,3)

web browser, are widely distributed and they are typically developed independently of the web service, it is critical that any fault tolerance scheme used be transparent to the client. Schemes for transparent server replication [3, 7, 18, 25] sometimes require deterministic servers for reply generation or do not recover requests whose processing was in progress at the time of failure. We discuss some of these solutions in more detail in Sections II and V.

We have previously developed a scheme for client-transparent fault-tolerant web service that overcomes the disadvantages of existing schemes [1]. The scheme is based on logging of HTTP requests and replies to a hot standby backup server. Our original implementation was based on user-level proxies, required non-standard features of the Solaris raw socket interface, and was never integrated with a real web server. That implementation did not require any kernel modifications but incurred high processing overhead. The contribution of this paper is a more efficient implementation of the scheme on Linux based on kernel modifications and its integration with the Apache web server using Apache's module mechanism. The small modifications to the kernel are used to provide client-transparent multicast of requests to a primary server and a backup server as well as the ability to continue transmission of a reply to the client despite server failure. Our implementation is based on off-the-shelf hardware (PC, router), and software (Linux, Apache). We rely on the standard reliability features of TCP and do not make any changes to the protocol or its implementation.

In Section II we present the architecture of our scheme and key design choices. Section III discusses our implementation based on kernel and web server modules. A detailed analysis of the performance results including throughput, latency, and consumed processing cycles is presented in Section IV. Related work is discussed in Section V.

II. TRANSPARENT FAULT-TOLERANT WEB SERVICE

In order to provide client-transparent fault-tolerant web service, a fault-free client must receive a valid reply for every request that is viewed by the client as having been delivered. Both the request and the reply may consist of multiple TCP packets. Once a request TCP packet has been acknowledged to the client, it must not be lost. All reply TCP packets sent to the client must form consistent, correct replies to prior requests.

We assume that only a single server host at a time may fail. We further assume that hosts are fail-stop [24]. Hence, host failure is detected using standard techniques, such as periodic heartbeats. Techniques for dealing with failure modes other than fail-stop are important but are beyond the scope of this paper. We also 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 [19]. This can prevent the local LAN router from being a critical single point of failure.

In order to achieve the fault tolerance goals, active replication of the servers may be used, where every client request is processed by both servers. While this approach will have the best fail-over time, it suffers from several drawbacks. First, this approach has a high cost in terms of processing power, as every client request is effectively processed twice. A second drawback is that this approach only works for deterministic servers. If the servers generate replies non-deterministically, the backup may not have an identical copy of a reply and thus it can not always continue the transmission of a reply should the primary fail in the midst of sending a reply.

An alternative approach is based on logging. Specifically, request packets are acknowledged only after they are stored redundantly (logged) so that they can be obtained even after a failure of a server host [1, 3]. Since the server may be non-deterministic, none of the packets of a reply can be sent to the client unless the entire reply is safely stored (logged) so that its transmission can proceed despite a failure of a server host [1]. The logging of requests can be done at the level of TCP packets [3] or at the level of HTTP requests [1]. If request logging is done at the level of HTTP requests, the requests can be matched with logged replies so that a request will never be reprocessed following failure if the reply has already been logged [1]. This is critical in order to ensure that for each request only one reply will reach the client. If request logging is done strictly at the level of TCP packets [3], it is possible for a request to be replayed to a spare server following failure despite the fact that a reply has already been sent to the client. Since the spare server may generate a *different* reply, two different replies for the same request may reach the client, clearly violating the requirement for transparent fault tolerance.

We have previously proposed [1] implementing transparent fault-tolerant web service using a hot standby backup server that logs HTTP requests and replies but does not actually process requests unless the primary server fails. The error control mechanisms of TCP are used to provide reliable multicast of client requests to the primary and backup. All client request packets are logged at the backup before arriving at the primary and the primary reliably forwards a copy of the reply to the backup before sending it to the client. Upon failure of the primary, the backup seamlessly takes over receiving partially received requests and transmitting logged replies. The backup processes logged requests for which no reply has been logged and any new requests.

Since our scheme is client-transparent, clients communicate with a single server address (the *advertised address*) and are unaware of server replication [1]. The backup server receives all the packets sent to the advertised address and forwards a copy to the primary server. For client transparency, the source addresses of all packets received by the client must be the advertised address. Hence, when the primary sends packets to the clients, it “spoofs” the source address, using the service’s advertised address instead of its own as the source address. The primary logs replies by sending them to the backup over a reliable (TCP) connection and waiting for an acknowledgment before sending them to the client. This paper uses the same basic scheme but the focus here is on the design and evaluation of a more efficient implementation based on kernel modifications.

III. IMPLEMENTATION

There are many different ways to implement the scheme described in Section II. As mentioned earlier, we have previously done this based on user-level proxies, without any kernel modifications [1]. A proxy-based implementation is simpler and potentially more portable than an implementation that requires kernel modification but it incurs higher performance overhead (Section IV). It is also possible to implement the scheme entirely in the kernel in order to minimize the overhead [22]. However it is generally desirable to minimize the complexity of the kernel [8, 17]. Furthermore, the more modular approach described in this paper makes it easier to port the implementation to other kernels or other web servers.

Our current implementation consists of a combination of kernel modifications and modifications to the user-level web server (Figure 2). TCP/IP packet operations are performed in the kernel and the HTTP message operations are performed in the web servers. We have not implemented the back-end portion of the three-tier structure. This can be done as a mirror image of the front-end communication [1]. Furthermore, since the transparency of the fault tolerance scheme is not critical between the web server and back-end servers, simpler and less costly schemes are possible for this section. 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

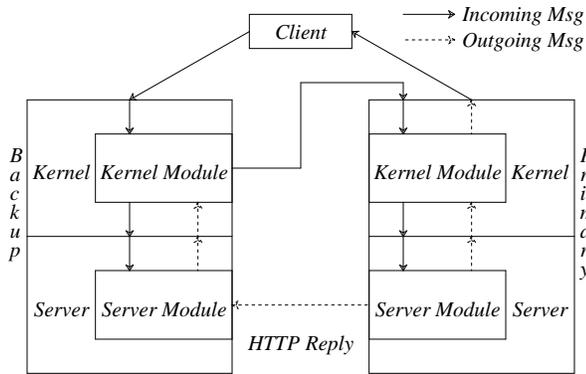


Figure 2: Implementation: replication using a combination of kernel and web server modules. Message paths are shown.

back-end can use that to avoid performing the transaction multiple times [20].

A. The Kernel Module

The kernel module implements the client-transparent atomic multicast mechanism between the client and the primary/backup server pair. In addition it facilitates the transmission of outgoing messages from the server pair to the client such that the backup can continue the transmission seamlessly if the primary fails.

The public address of the service known to clients is mapped to the backup server, so the backup will receive the client packets. After an incoming packet goes through the standard kernel operations such as checksum checking, and just before the TCP state change operations are performed, the backup's kernel module forwards a copy of the packet to the primary. The backup's kernel then continues the standard processing of the packet, as does the primary's kernel with the forwarded packet.

Outgoing packets to the client are sent by the primary. Such packets must be presented to the client with the service public address as the source address. Hence, the primary's kernel module changes the source address of outgoing packets to the public address of the service. On the backup, the kernel processes the outgoing packet and updates the kernel's TCP state, but the kernel module intercepts and drops the packet when it reaches the device queue. TCP acknowledgments for outgoing packets are, of course, incoming packets and they are multicast to the primary and backup as above.

The key to our multicast implementation is that when the primary receives a packet, it is assured that the backup has an identical copy of the packet. The backup forwards a packet only *after* the packet passes through the kernel code where a packet may be dropped due to a detected error (e.g., checksum) or heavy load. If a forwarded packet is lost while enroute to the primary, the client does not receive an acknowledgment and thus retransmits the packet. This is because only the primary's TCP acknowledgments reach the client. TCP acknowledgments generated by the backup are dropped by the backup's kernel module.

B. The Server Module

The server module is used to handle the parts of the scheme that deal with messages at the HTTP level. The Apache module acts as a handler [27] and generates the replies that are sent to the clients. It is composed of worker, mux, and demux processes.

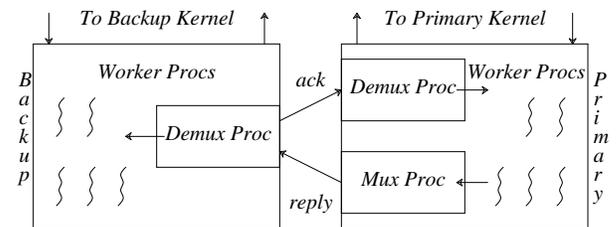


Figure 3: Server Structure: The mux/demux processes are used to reliably transmit a copy of the replies to the backup before they are sent to clients. The server module implements these processes and the necessary changes to the standard worker processes.

1) *Worker Processes:* A standard Apache web server consists of several processes handling client requests. We refer to these standard processes as worker processes. In addition to the standard handling of requests, in our scheme the worker processes also communicate with the mux/demux processes described in the next subsection.

The primary worker processes receive the client request, perform parsing and other standard operations, and then generate the reply. Other than a few new bookkeeping operations, these operations are exactly what is done in a standard web server. After generating the reply, instead of sending the reply directly to the client, the primary worker processes pass the generated reply to the primary mux process so that it can be sent to the backup. The primary worker process then waits for an indication from the primary demux process that an acknowledgment has been received from the backup, signaling that it can now send the reply to the client.

The backup worker processes perform the standard operations for receiving a request, but do not generate the reply. Upon receiving a request and performing the standard operations, the worker process just waits for a reply from the backup demux process. This is the reply that is produced by a primary worker process for the same client request.

2) *Mux/Demux Processes:* The mux/demux processes ensure that a copy of the reply generated by the primary is sent to and received by the backup before the transmission of the reply to the client starts. This allows for the backup to seamlessly take over for the primary in the event of a failure, even if the replies are generated non-deterministically. The mux/demux processes communicate with each other over a TCP connection, and use semaphores and shared memory to communicate with worker processes on the same host (figure 3). A connection identifier (client's IP address and TCP port number) is sent along with the replies and acknowledgments so that the demux process on the remote host can identify the worker process with the matching request.

IV. PERFORMANCE EVALUATION

The evaluation of the scheme was done on 350 MHz Intel Pentium II PC's interconnected by a 100Mb/sec switched network based on a Cisco 6509 switch. The servers were running our modified Linux 2.4.2 kernel and the Apache 1.3.23 web server with logging turned on and with our kernel and server modules installed. We used custom clients similar to those of the Wisconsin Proxy Benchmark [2] for our measurements. The clients continuously generate one outstanding HTTP request at a time with no think time. For each experiment, the requests were for files of a specific size as presented in our results. Internet traffic studies [13, 10] indicate that most web replies are less than 10-15 kbytes in size. Measurement were conducted on at least three system configurations: unreplicated, simplex, and duplex. The "unreplicated" system is the standard system with no kernel or web server modifications. The "simplex" system includes the kernel and server modifications but there is only one server, i.e., incoming packets are not really multicast and outgoing packets are not sent to a backup before transmission to the client. The extra overhead of "simplex" relative to "unreplicated" is due mainly to the packet header manipulations and bookkeeping in the kernel module. The "duplex" system is the full implementation of the scheme.

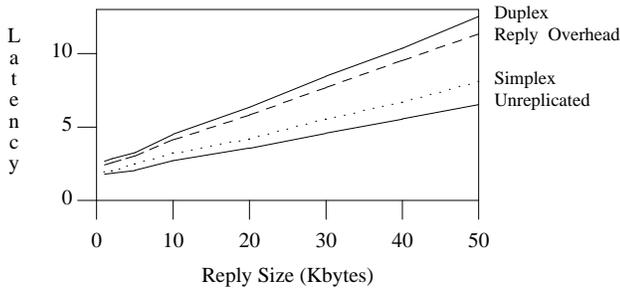


Figure 4: Average latency (msec) observed by a client for different reply sizes and system modes. The *Reply Overhead* line depicts the latency caused by replication of the reply in duplex mode.

A. Latency

Figure 4 shows the average latency on an unloaded server and network from the transmission of a request by the client to the receipt of the corresponding reply by the client. There is only a single client on the network and this client has a maximum of one outstanding request. The results show that the latency overhead relative to the unreplicated system increases with increasing reply size. This is due to processing of more reply packets. The difference between the "Reply Overhead" line and the "Unreplicated" line is the time to transmit the reply from the primary to the backup and receive an acknowledgement at the primary. This time accounts for most of the duplex overhead. Note that these measurements exaggerate the relative overhead that would impact a real system since: 1) the client is on the same local network as the server, and 2) the requests are for (cached) static files. 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 our scheme remains the same regardless of server response times and therefore our implementation overhead is only a small fraction of the overall response time seen by clients.

B. Throughput

Figure 5 shows the peak throughput of a single pair of server hosts for different reply sizes. The throughputs of "unreplicated" and "simplex" (in Mbytes/sec) increase until the network becomes the bottleneck. However, the duplex mode throughput peaks at less than half of that amount. This is due to the fact that on the primary, the sending of the reply to the backup by the server module and the sending of reply to the clients (figure 2) occur over the same physical link. Hence, the throughput to the clients is reduced by half in duplex mode. To avoid this bottleneck, the transmission of the replies from the primary to the backup can be performed on a separate dedicated link. A high-speed Myrinet [9] LAN was available to us and was used for this purpose in measurements denoted by "duplex-mi". These measurements show a significant throughput improvement over the duplex results, as a throughput of about twice that of duplex mode with a single network interface is achieved.

C. Processing Overhead

Table 1 shows the CPU cycles used by the servers to receive one request and generate a reply. These measurements were done using the processor's performance monitoring counters [21]. For each configuration the table presents the kernel-level, user-level, and total cycles used. The *cpu%* column shows the *cpu* utilization at peak throughput, and indicates that the system becomes *CPU* bound as the reply size decreases. This explains the throughput results, where lower throughputs (in Mbytes/sec) were reached with smaller replies.

Based on Table 1, the duplex server (primary and backup combined) can require more than four times (for the 50KB reply) as many cycles to handle a request compared with the unreplicated server. However, as noted in the previous subsection, these measurements are for replies generated by reading cached static files. In practice, for likely applications of this technology (dynamic content), replies are likely to be smaller and require significantly more processing. Hence, the actual relative processing overhead can be expected to be much lower than the factor of 4 shown in the table.

D. Comparison with a User-Level Implementation

As mentioned earlier, our original implementation of this fault tolerance scheme was based on user-level proxies, without any kernel modifications [1]. Table 2 shows a comparison of the processing overhead of the user-level proxy approach with the implementation presented in this paper. This comparison is not perfectly accurate. While both schemes were implemented on the same hardware, the user-level proxy approach runs under the Solaris operating system and could not be easily ported to Linux due to a difference in

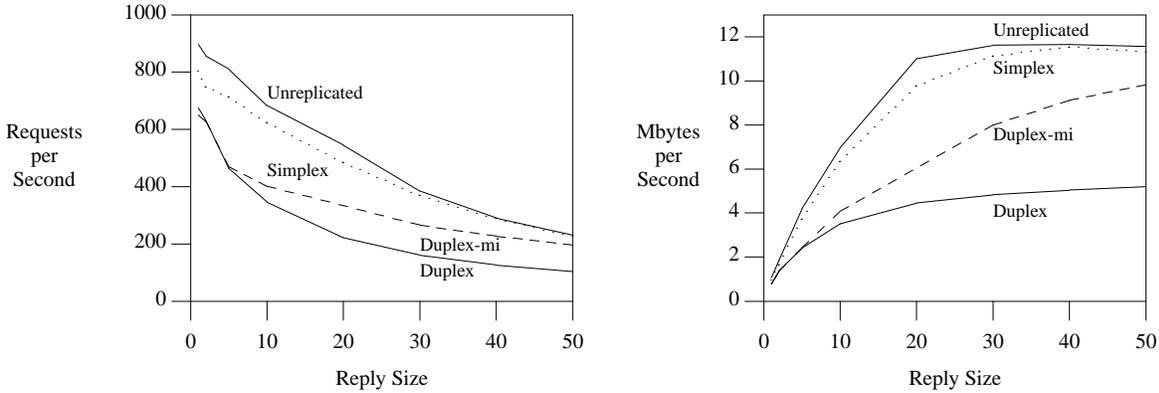


Figure 5: System throughput (in requests and Mbytes per second) for different message sizes (kbytes) and system modes. Duplex-mi line denotes setting with multiple network interfaces for each server - one interface is used only for reply replication.

TABLE 1: Breakdown of used CPU cycles (in thousands) - cpu% column indicates CPU utilization during peak throughput.

System Mode	1kbyte reply				10kbyte reply				50kbyte reply			
	user	kernel	total	cpu%	user	kernel	total	cpu%	user	kernel	total	cpu%
Duplex (primary)	190	337	527	100	193	587	780	77	224	1548	1772	53
Duplex (backup)	147	330	477	91	158	615	773	76	185	1790	1958	58
Duplex-mi (primary)	192	353	545	100	198	544	742	85	225	1283	1508	85
Duplex-mi (backup)	147	355	502	93	152	545	697	80	169	1124	1293	72
Simplex	186	250	436	100	191	365	556	99	208	871	1079	70
Unreplicated	165	230	395	100	166	342	508	99	178	730	908	60

TABLE 2: User-level versus kernel support — CPU cycles (in thousands) for processing a request that generates a 1Kbyte reply.

Implementation	Primary	Backup	Total
User-level Proxies	1860	1370	3230
Kernel/Server Modules	337	330	667

the semantics of raw sockets. In addition, the server programs are different although they do similar processing. However, the difference of almost a factor of 5 is clearly due mostly to the difference in the implementation of the scheme, not to OS differences. The large overhead of the proxy approach is caused by the extraneous system calls and message copying that are necessary for moving the messages between the two levels of proxies and the server.

V. RELATED WORK

Early work in this field, such as Round Robin DNS [11] and DNS aliasing methods, focused on detecting a fault and routing future requests to available servers. Centralized schemes, such as the Magic Router [4] and Cisco Local Director [12], 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 [7] 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. None of these schemes support recovering requests that were being processed when the failure occurred, nor do they deal with non-deterministic and non-idempotent requests.

There are various server replication schemes that are not client transparent. Most still do not provide recovery of requests that were partially processed. Frolund and Guerraoui [16] do 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 kernel module can be seen as an alternative implementation of the write-once registers which also provides client transparency. Zhao et al [29] describe a CORBA-based infrastructure for replication in three-tier systems which deal with the same issues, but again is not client-transparent.

The work by Snoeren et al [26] is another example of a solution that is not transparent to the client. A transport layer protocol with connection migration capabilities, such as SCTP or TCP with proposed extensions, is used along with a session state synchronization mechanism between servers to achieve connection-level failover. The requirement to use a specialized transport layer protocol at the client is obviously not transparent to the client.

HydraNet-FT [25] 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 [1]. 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 active replication, it can only be used with deterministic servers while our standby backup scheme does not have this limitation.

Alvisi et al implemented FT-TCP[3], a kernel level TCP wrapper that transparently masks server failures from clients. While this scheme and its implementation are similar to ours, there are important differences. Instead of our hot standby spare approach, a logger running on a separate processor is used. If used for web service fault tolerance, FT-TCP requires deterministic servers (see Section II) and significantly longer recovery times. In addition, they did not evaluate their scheme in the context of web servers.

VI. CONCLUSION

We have proposed a client-transparent fault tolerance scheme for web services that correctly handles all client requests in spite of a web server failure. Our scheme is compatible with existing three-tier architectures and can work with non-deterministic and non-idempotent servers. We have implemented the scheme using a combination of Linux kernel modifications and modification to the Apache web server. We have shown that this implementation involves significantly lower overhead than a strictly user-level proxy-based implementation of the same scheme. Our evaluation of the response time (latency) and processing overhead shows that the scheme does introduce significant overhead compared to a standard server with no fault tolerance features. However, this result only holds if generating the reply requires almost no processing. In practice, for the target application of this scheme, replies are often small and are dynamically generated (requiring significant processing). For such workloads, our results imply low relative overheads in terms of both latency and processing cycles. We have also shown that in order to achieve maximum throughput it is critical to have a dedicated network connection between the primary and backup.

REFERENCES

- [1] N. Aghdaie and Y. Tamir, "Client-Transparent Fault-Tolerant Web Service," *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, pp. 209-216 (April 2001).
- [2] J. Almeida and P. Cao, "Wisconsin Proxy Benchmark," *Technical Report 1373, Computer Sciences Dept, Univ. of Wisconsin-Madison* (April 1998).
- [3] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping Server-Side TCP to Mask Connection Failures," *Proceedings of IEEE INFOCOM*, Anchorage, Alaska, pp. 329-337 (April 2001).
- [4] E. Anderson, D. Patterson, and E. Brewer, "The Magicrouter, an Application of Fast Packet Interposing," *Class Report, UC Berkeley* - <http://www.cs.berkeley.edu/~eanders/projects/magicrouter/> (May 1996).
- [5] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers," *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, pp. 850-856 (April 1996).
- [6] S. M. Baker and B. Moon, "Distributed Cooperative Web Servers," *The Eighth International World Wide Web Conference*, Toronto, Canada, pp. 1215-1229 (May 1999).

- [7] A. Bestavros, M. Crovella, J. Liu, and D. Martin, "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).
- [8] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, and R. P. Draves, "Microkernel Operating System Architecture and Mach," *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Berkeley, CA, pp. 11-30 (April 1992).
- [9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro* **15**(1), pp. 29-36 (February 1995).
- [10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *Proceedings of IEEE INFOCOM*, New York, New York (March 1999).
- [11] T. Brisco, "DNS Support for Load Balancing," *IETF RFC 1794* (April 1995).
- [12] 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.
- [13] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of World Wide Web Client-based Traces," *Technical Report TR-95-010, Boston University, CS Dept, Boston, MA 02215* (April 1995).
- [14] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A scalable and highly available web server," *Proceedings of IEEE COMPCON '96*, San Jose, California, pp. 85-92 (1996).
- [15] S. Frolund and R. Guerraoui, "CORBA Fault-Tolerance: why it does not add up," *Proceedings of the IEEE Workshop on Future Trends of Distributed Systems* (December 1999).
- [16] 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).
- [17] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program," *Proceedings of summer USENIX*, pp. 87-96 (June 1990).
- [18] 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).
- [19] S. Knight, D. Weaver, D. Whipple, R. Hinden, D. Mitzel, P. Hunt, P. Higginson, M. Shand, and A. Lindem, "Virtual Router Redundancy Protocol," RFC 2338, IETF (April 1998).
- [20] Oracle Inc, *Oracle8i Distributed Database Systems - Release 8.1.5*, Oracle Documentation Library (1999).
- [21] M. Pettersson, "Linux x86 Performance-Monitoring Counters Driver," <http://www.csd.uu.se/~mikpe/linux/perfctr/>.
- [22] Red Hat Inc, "TUX Web Server," <http://www.redhat.com/docs/manuals/tux/>.
- [23] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek, "Selection Algorithms for Replicated Web Servers," *Performance Evaluation Review - Workshop on Internet Server Performance*, Madison, Wisconsin, pp. 44-50 (June 1998).
- [24] F. B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).
- [25] G. Shenoy, S. K. Satapati, and R. Bettati, "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).
- [26] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, "Fine-Grained Failover Using Connection Migration," *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California (March 2001).
- [27] L. Stein and D. MacEachern, *Writing Apache Modules with Perl and C*, O'Reilly and Associates (March 1999).
- [28] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann, "Web++: A System For Fast and Reliable Web Service," *Proceedings of the USENIX Annual Technical Conference*, Sydney, Australia, pp. 171-184 (June 1999).
- [29] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, "Increasing the Reliability of Three-Tier Applications," *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Hong Kong, pp. 138-147 (November 2001).