Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing
Victoria, Canada, pp. 29-32, August 2003.

Performance Optimizations for Transparent Fault-Tolerant Web Service

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

***Abstract*—Reliable Web service requires the ability to complete transactions that are in progress when a Web server fails. We have previously presented a client-transparent scheme, based on a standby backup and logging, for providing such fault-tolerant Web service. The scheme does not require deterministic servers and can thus properly handle dynamic content. This paper presents two performance optimizations that significantly reduce the overhead of the scheme. For dynamic content, the throughput of a server cluster is increased by distributing the primary and backup tasks among the servers. For static content, that is deterministic and readily generated, the overhead is reduced by avoiding explicit logging of replies to the backup. Our implementation is based on special modules in the Linux kernel and the Apache Web server. We discuss implementation issues and present overhead measurements in terms of latency, throughput, and CPU cycles.**

I. Introduction

Web service for critical applications is often provided by 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). We have previously proposed and implemented a client-transparent fault-tolerant Web service scheme based on a hot standby backup front-end server and logging [1, 2]. Our scheme recovers in-progress requests and does not require deterministic servers or changes to the clients. In this paper, we describe and evaluate the implementation of two performance enhancements. In our original implementation each node in a primary/backup pair maintained its role as the primary or the backup for all requests for as long as both nodes were operational. This results in unbalanced utilization of the nodes. The first performance enhancement we propose is to allow each node to serve as primary for some requests and backup for others. Our original scheme was focused on handling requests for dynamic content so that the replies generated may not be deterministic. With requests for static content, this approach has unnecessarily high overhead. The second performance enhancement we propose here is to handle requests for dynamic content and static content differently, requiring lower overhead for the latter requests.

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 by a server, it must not be lost. All reply TCP packets sent to the client must form consistent, correct replies to prior requests.

Our scheme logs HTTP requests and replies to a hot standby backup (Figure 1) [1, 2]. Clients are unaware of the duplex servers and communicate with a single IP address. 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. Since the server may be non-deterministic, none of the reply packets are sent to the client unless the entire reply is safely stored (logged) so that its transmission can proceed despite a server host failure. Since the request logging is done at the HTTP level, 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. This is critical in order to ensure that for each request only one reply will reach the client.

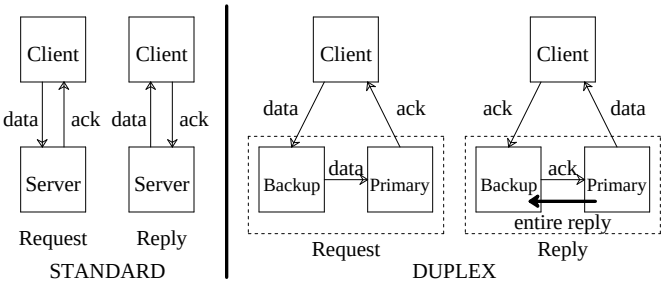

Figure 1: Message paths for a standard unreplicated server and a hot standby replication scheme. Replicated servers appear as a single entity to clients. Each client request is logged by the backup before being forwarded to the primary. The reply is generated by the primary and reliably sent to the backup and logged before being sent to the client.

Our more efficient implementation [2], which is the starting point for this paper, utilizes a Linux kernel module and an Apache Web server module. Clients send their requests to the advertised service IP address, which is mapped to the backup server during normal operation. The backup kernel module forwards a copy of the packets to the primary after they have been logged, effectively multicasting the requests. The primary server module processes the request and generates a reply. It then sends the reply to the backup over a reliable connection for logging, and waits for an acknowledgment from the backup server module. The server modules on both the primary and backup then pass the replies to their respective kernels. The kernel modules provide the transparent reply transmission by assuring that only one copy of the reply is sent to the client, while preserving the connection state. The primary kernel module sends the reply to client, using the advertised address as the source address of the packets. The backup kernel module discards the messages as it receives acknowledgments from the client for messages sent by the primary. Further details of the scheme, including the fault model and rationale for design choices, are explained in our previous papers [1, 2].

III. Two Performance Enhancements

We have implemented two performance optimizations for our original scheme. The first, improves throughput by distributing the primary and backup tasks among all the server hosts. The second, reduces the average overhead per request by allowing a more efficient scheme to be used with requests for static content.

A. Dual-Role Server Hosts

Using our original scheme, when serving dynamic content that requires significant processing in order to generate each reply, the primary is likely to require significantly more processing than the logging that is performed on the backup. This type of content is common for transaction processing e-commerce applications, where fault-tolerance is critical. For the experimental setup described in Section IV, Figure 2 shows the CPU cycles used per reply, where replies are generated using the WebStone [6, 11] CGI benchmark. The large difference between the processing cycles used at the primary and backup servers indicate that the backup host is mostly idle, with its processing potential largely wasted. A simple solution to this problem is to distribute the primary server tasks and backup server tasks among all the hosts. Hence, each server host will serve as the primary for some requests and the backup for others [3, 10]. We refer to this scheme as *dual-role servers*. The distribution of requests can be done using standard load balancing techniques, such as Round Robin DNS [4] or a centralized load balancer (with failover capability) [5].

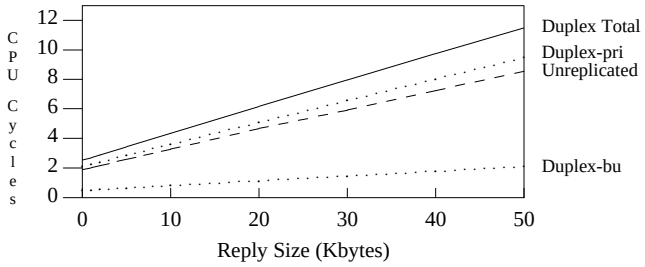

Figure 2: Server hosts CPU cycles (in millions) per request for processing requests requiring dynamically-generated replies of different sizes. The primary and backup nodes of the system in duplex mode are denoted by *Duplex-pri* and *Duplex-bu*, respectively. The *Duplex Total* line is the sum of the cycles used by the primary and backup per request.

In our original implementation, the kernel modules and the Apache server modules were statically initialized to perform the primary or backup functions [2]. For the dual-role server optimization, the kernel module must perform both functions simultaneously. Hence, for each packet received, the module must dynamically determine whether to invoke primary or backup processing. In addition, the Web server processes (server modules) need to dynamically determine whether to perform primary or backup processing at the HTTP level, These requirements are met using two separate TCP ports: one for incoming packets from clients and another for the forwarded packets from the other server. When a kernel module receives a packet on the public (client) port, it functions as the backup and forwards a copy of the packet to the internal (forwarding) port of the other server. Based on the port number through which the packet arrives, the kernel module performs the appropriate processing. The same basic mechanism also works for the server module. Although user-level server modules do not have access to packet headers, they can determine the message destination address and the appropriate mode based on the socket through which the request is received.

B. Efficient Handling of Static, Deterministic Content

Our replication scheme was designed to handle non-deterministic dynamic reply generation. The reply is generated by the primary and then logged on the backup before its transmission to the client can begin. The logging is done over a dedicated TCP connection between the primary and backup. The primary waits for an explicit user-level acknowledgment from the backup before it begins to transmit the reply to the client [1, 2]. Compared with transmission of the reply as soon as it is generated, our scheme results in increased latency. Specifically, most of the latency overhead of our scheme is due to the logging of the replies (see Section IV) and increases with message size.

Much of the latency overhead of our scheme can be eliminated if the replies are deterministic — for example, if the replies are based on the contents of static data (files) available to all hosts. In that case, instead of logging replies, active replication [7, 8] can be used, where both the primary and backup independently generate each reply.

A possible disadvantage of using active replication is increased CPU load on the backup for generating the replies. However, deterministic replies of Web services are often generated from static files and their generation is not processor intensive. Even processor intensive deterministic server applications (e.g. deterministic CGI scripts), often have their results pre-computed and preserved in ‘‘cache’’ files for performance reasons. With reply logging, for replies generated from cached files, the number of CPU cycles required by the backup server for logging the replies is approximately the same as the number of CPU cycles required by the primary to generate the replies [2]. Thus, for cached static files, the total number of CPU cycles per request with active replication can be expected to be lower than with reply logging since the primary server’s CPU cycles for transmitting the reply to the backup are eliminated.

The lack of synchronization between the primary and backup servers with this optimization can cause a performance problem. Since replies are not logged and reply messages are not exchanged between servers, the backup may fall behind the primary. The primary server may process a request, produce the reply, and send the reply to the client all *before* the backup server processes the same request. In such a case, the backup receives client TCP acknowledgment packets *before* it has produced the corresponding TCP data packets. The backup kernel module drops these acknowledgments, allowing the primary and backup TCP states to converge before the acknowledgment packets are processed. This approach can lead to retransmission of some packets and an increase in the observed request processing time by some of the clients. This problem can be fairly common under heavy load because both servers perform identical operations except that the backup performs the extra step of forwarding every client packet to the primary, making the backup the processing bottleneck of the system.

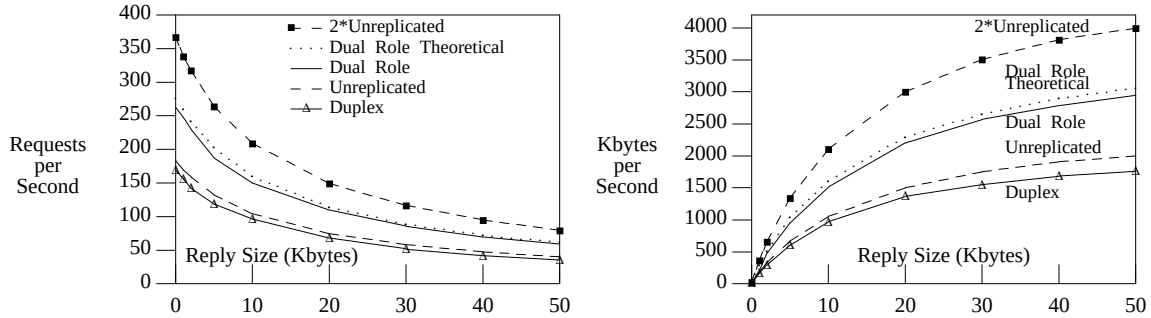

Figure 3: Dual role optimization: peak throughput for different reply message sizes Replies generated with WebStone 2.0 CGI benchmark.

The problem with packet retransmission described above can be alleviated, at a cost of some latency and processing overhead, using a scheme we call *sync static* (synchronized static). With the sync static approach, the backup server sends a message containing the connection identifier to the primary upon the generation of each reply. The primary sends the reply to the client only after it receives the synchronization message from the backup. This prevents the primary from getting too far ahead of the backup.

The active replication version of our scheme and the original reply logging version of our scheme can be used simultaneously on the same servers. The Apache Web server provides a mechanism similar to a content-based (layer-7) router, where decisions about the processing of a request can be made based on the request URL in the HTTP header. Instead of routing requests to different servers as done by routers, Apache decides whether or not to use each module based on the request URL [9]. If the request URL includes a path that has been designated to be non-deterministic content, our module that implements the reply logging is used. Otherwise, the module and the reply logging step is skipped. For example, servers can be setup where the URLs for non-deterministic content begin with *http://hostname/non-deterministic/*, and those for deterministic or static content begin with *http://hostname/static/*.

A conceivable further optimization is to have the backup server log the requests and generate the reply *only* if the primary fails. As a result, processing cycles on the backup would not have to be used for reply generation during normal operation. Our original non-deterministic scheme also accomplishes this goal with the additional cost of sending the replies from the primary to the backup. Hence, the benefits of this possible approach are limited to only the cases where the replies are deterministic, processor intensive, and very large.

IV. 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. The dynamic server replies were generated using WebStone 2.0 benchmark's CGI workload generator [6, 11]. This benchmark randomly generates each reply byte. The static replies were generated from files that were cached in the server's memory. We used custom clients that continuously generate one outstanding HTTP request at a time. For each experiment, all the replies were the same size. We compared the results with the performance optimizations to the results of two other configurations: the *unreplicated* system — the standard simplex server with no kernel or Web server modifications, and the *duplex* system — our previous implementation of fault-tolerant Web service [2].

A. Dual-Role Servers

The impact of the dual-role optimization on peak throughput is shown in Figure 3. To emulate the operation of a load-balancing mechanism that distributes requests among multiple servers [5], the clients were manually configured so that half of the requests were sent to each server. Our original *duplex* system, using two server hosts, achieves almost the same throughput as a single host running the *unreplicated* server. Hence, using the same amount of resources (two server hosts), the *unreplicated* system can achieve slightly more than twice the throughput. Specifically, the difference between the *2*unreplicated* line and the *duplex* line represents the throughput overhead of our original scheme.

The *dual role* results show significant improvements over our original implementation. As mentioned earlier, the performance improvement is due to the use of otherwise idle backup cycles for processing of requests. The *dual role theoretical* line shows the theoretical upper bound for our optimized throughput. The values were calculated using measurements of required CPU cycles for the processing of a single request and the known processing speed of our servers. The small difference between the *dual role theoretical* and *dual role* lines are likely due to the increased number of context switches that occur during high loads.

B. Efficient Handling of Static, Deterministic Content

The impact of the optimizations for static content on peak throughput is shown in Figure 4. Replies are generated from cached static files. For large replies, the throughput difference between an unreplicated server and our original *duplex* scheme is largely due to the network becoming a throughput bottleneck due to the logging of replies [2]. We have previously shown that this situation can be greatly improved by using two network interfaces on each server, with one being dedicated for the reply logging [2]. The *duplex-mi* line shows the benefit of using such a dedicated connection. The optimizations for static content (*static* and *sync static*) eliminate reply logging and thus, as shown in Figure 4, exceeds the performance of the *duplex-mi* scheme, without using a dedicated network connection.

Figure 5 shows the average request latency as observed by a client for lightly loaded servers. Specifically, only a single client accessing the servers, one request at a time. The difference between the *Reply Overhead* and *Unreplicated* lines is the latency overhead for reply logging, which accounts

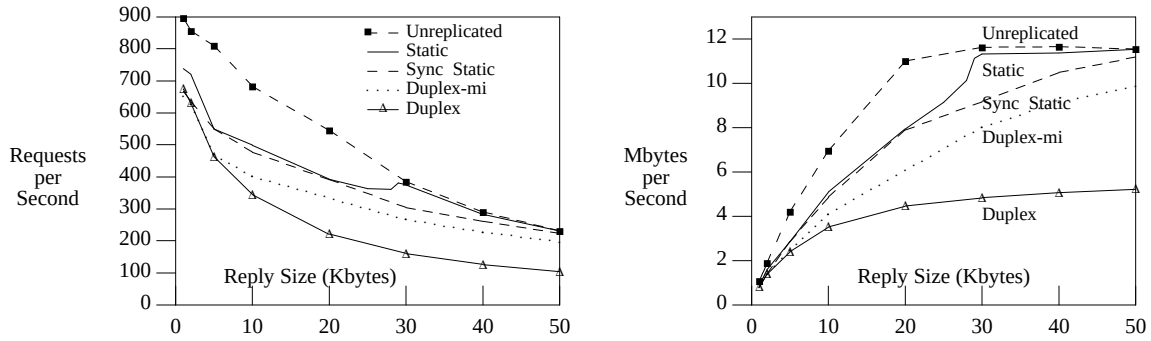

Figure 4: Optimizations for static content: peak system throughput for different reply sizes. Replies generated from cached static files. The *Duplex-mi* line is for a duplex system with a dedicated network between the servers. The *Static* and *Sync Static* lines are for the static content optimizations.

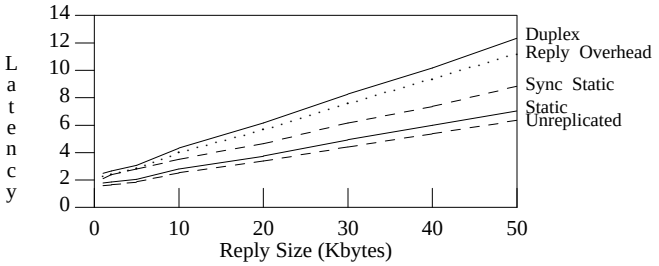

Figure 5: Average latency (msec) observed by a client for dynamically generated replies of different sizes. The *Reply Overhead* line shows the fraction of *Duplex* mode overhead due to reply logging. The *Static* and *Sync Static* lines show the latency with the two alternative optimizations for static content.

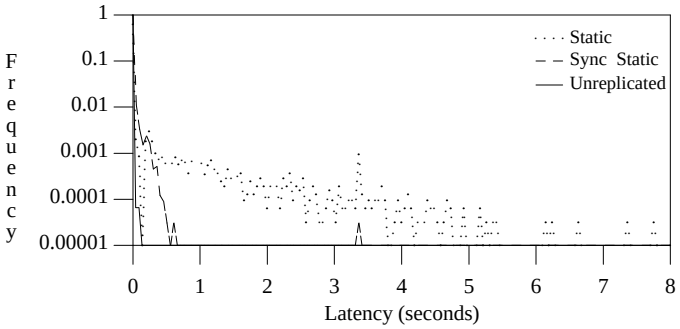

Figure 6: Latency distribution for 5 Kbyte static replies at 550 requests/second throughput. (Note the log scale on the Y axis).

for most of the latency overhead of the duplex scheme. Thus, our optimizations for static replies, which eliminate the reply logging, result in a significant reduction of the latency overhead. In particular, the *static* line shows very little latency overhead compared to the *unreplicated* server.

Figure 6 shows the distribution of request-reply latencies for the *unreplicated*, *static*, and *sync static* schemes for 5Kbytes static replies and a throughput of 550 requests per second. While the overwhelming majority of requests are processed within a few milliseconds, with the *static* optimization, under heavy load, a tiny fraction of requests result in latencies on the order of few seconds. This is due to an increased probability of required retransmissions, as discussed in Section III. The figure also shows that the *sync static* optimization dramatically reduces the probability of long latency relative to the *static* scheme. Figures 4 and 5 show that the reduced latency variance of the *sync static* scheme compared to the *static* scheme comes at the cost of reduced peak throughput as well as increased average latency.

V. Conclusion

We have proposed, implemented, and evaluated two techniques for improving the performance of our transparent fault-tolerant Web service. The dual-role server optimization ensures efficient utilization of available processing resources. For large messages (50KB), we have demonstrated a throughput improvement of up to 67% compared to a scheme where each server host must be dedicated to a single role: primary or backup. The optimizations for static content allow the server to simultaneously and efficiently serve both static and dynamic content, using the best-suited scheme for each request type. For large messages (50KB), we have demonstrated a request-reply average latency reduction of up to 43% coupled with a throughput increase of up to 121% compared to the scheme based on reply logging, as needed for non-deterministic content.

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] A. Borg, J. Baumbach, and S. Glazer, ''A Message System Supporting Fault Tolerance,'' *9th Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).

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

[5] 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.*

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

[7] F. B. Schneider, ''Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,'' *ACM Computing Surveys*, pp. 299-319 (December 1990).

[8] G. Shenoy, S. K. Satapati, and R. Bettati, ''HydraNet-FT: Network Support for Dependable Services,'' *20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 699-706 (April 2000).

[9] L. Stein and D. MacEachern, *Writing Apache Modules with Perl and C*, O'Reilly and Associates (March 1999).

[10] TimesTen Inc, ''Data Replication and TimesTen,'' *http://www.timesten.com* (2002).

[11] G. Trent and M. Sake, ''WebSTONE: The First Generation in HTTP Server Benchmarking,'' *http://www.mindcraft.com/webstone /paper.html* (February 1995).