Constructing Zero-Loss Web Services

Mon-Yen Luo and Chu-Sing Yang

Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C

Abstract-- Modern Web services must support large and rapidly growing user populations, and remain available 24 hours per day, 7 day per week. Server cluster is the most promising approach to address this challenge. In this paper, we further augment the server-cluster schemes with a novel mechanism that enables a Web request to be smoothly migrated and recovered on another working node in the presence of server failure or overload. The new mechanism provides a powerful solution to fault tolerance and dynamic load-distribution for Web services. The administrator can explicitly specify some services to be guaranteed for fault-tolerance or higher performance support. We present the details of our design, implementation, and performance data. The performance results show that the proposed mechanism is efficient and with low associated overhead.

I. INTRODUCTION

As the Web has been a vital infrastructure for a variety of commercial services, service credibility of the Web server becomes a great concern. In particular, the modern Web server must cope with many challenging problems than ever before. The Web services have become more sophisticated as the Web continues to evolve from its initial role as a provider of read-only documentation-based information to a platform for supporting complex services. The Web server must be able to service thousands of simultaneous client requests and scale to rapidly growing user populations. As a result, the credibility of a Web site usually suffers from two problems: server failure and server overload. Rapid response and high reliability are mandatory requirements for a Web site while service outage in today's highly competitive marketplace can mean lost revenue and lost credibility.

Server replication (or server clustering [1]) may be the most promising solution to the above challenges. Over the past few years, numerous server-clustering schemes have been proposed. Examples include client-side approach [2], DNS aliasing [3], TCP connection routing [4,5,6], HTTP redirection [7], and others [8,9]. These solutions indeed can provide compelling performance and scalability. However, we find that they merely can provide high availability via its redundant nature, but offer no guarantee about fault resilience for the service. In these clustered server systems, when one server node fails, the failure can be detected and transparently replaced with the available redundant server, resulting in high availability to the services. However, any ongoing requests on the failed server will be lost. In addition to detecting and masking the failures, an ideal highly reliable server should enable the outstanding requests on the failed node to be smoothly migrated and then recovered on another working node.

The second flaw in much of the server cluster is the lack of adaptability to surge load bursts. In addition to the rapidly increasing client population, the Web is also well known for having highly varied load. Certain event could trigger a significant load burst that persists for hours, or even days. Examples include announcement of a new version of popular software or products, or a site mentioned as the “best-site-of-the-week” on the news. As a result, some nodes in the cluster could suddenly become swamped due to receiving far more requests than it was originally configured to handle. This problem is particularly troubling for sites that offer E-commerce services, since the requests for popular pages may overwhelm the requests for more important services (e.g., merchant services or services for higher-paying customers). The requests for these critical services should remain available even when the server is heavily loaded, or take higher priority over other trivial requests.

The above problems motivated us to devise a mechanism that can enable a Web request to be smoothly migrated and recovered on another working node in the presence of server failure or overload. Our goal is to provide a zero-loss web service, that is, our system will guarantee the service of all user submitted requests against any loss. In this paper, we focus attention on the viewpoint of server side. We do not cover the service outage caused by network congestion or failure of the path from the client to the server. We leave this issue to future work.

In the rest of this paper, we will describe the design, implementation and performance of the proposed mechanism. Given a clustered server, some request-routing mechanism is needed to dispatch and route the incoming request to the server best suited to respond. To effectively support zero-loss web services, we augment the request-routing mechanism with two new capabilities: request discrimination and request migration. The administrator can explicitly specify that some services should be guaranteed for fault-tolerance support or higher performance. We describe the request-discrimination mechanism in section 2. In section 3, we will describe the request-migration mechanism, which could transparently enable a request to be migrated and recovered in a Web-server cluster. Section 4 discusses some related issues of our system. We present the results of performance
evaluation on the prototype system in section 5. The performance results show that the proposed mechanism is efficient and with low associated overhead. We discuss related work in section 6, and then draw conclusions in section 7.

II. REQUEST DISCRIMINATION

In this section, we first analyze why the request discrimination is needed. We then describe our design and implementation to realize such a new capability.

A. Analysis

To provide zero-loss Web services, we think the routing mechanism in the server cluster should be extended to support two important capabilities: checkpointing and recovery. That is, some intermediate state of user requests should be logged by the checkpointing mechanism. When one server fails, the recovery mechanism can enable the outstanding requests on the failed (or overloaded) node to be continued processing with a valid intermediate state in another working node. Although the two techniques have been well investigated in the research area of fault tolerance, implementing these techniques in the web server cluster still pose many new challenges.

First, the cost is very expensive if we log every incoming request for checkpointing. In current Web sites, the service type of incoming requests can be as varied as static Web pages, dynamic content generated by CGI scripts, or transaction-based services. It is obvious that not every request is necessary to be logged for checkpointing. As a result, we argue that the routing mechanism in the Web server cluster should be “content-aware” so that it can differentiate some mission-critical requests from regular Web surfing requests. Second, how to recover a Web request from a failed server node to continue execution in another working node is a challenging problem. In particular, such a recovery mechanism should be user-transparent and smooth. Motivated by these observations, we designed our request routing mechanism to address these problems as follow.

B. Our design

The request routing mechanism of our system is based on our previous work [10]. We briefly describe the operation of the previous mechanism as follows. The dispatcher node that executes the routing mechanism will pre-fork a number of persistent connections [11] to the back-end nodes, and then allocates system resources by dispatching client requests on these trunks. When a client tries to retrieve specific content, the client-side browser first needs to create a TCP connection. The incoming TCP connection requests are acknowledged and handled at the dispatcher until the client sends packets conveying the HTTP request, which contains the URL (specifies the specific content it is asking for) and other HTTP client header information (e.g., Host, cookie, etc.). At that point, the dispatcher looks into the HTTP header to make decision on how to route the request. When the dispatcher selects a server that is best suited to this request, it then chooses an idle pre-forked connection from the available connection list of the target server. The dispatcher then stores related information (e.g., TCP states) about the selected connection in an internal data structure termed “mapping table”, binding the user connection to the pre-forked connection. After the connection binding is determined, the dispatcher handles the consequent packets by changing each packet’s IP and TCP headers for seamlessly relaying the packet between the user connection and the pre-forked connection, so that the server can transparently receive and recognize these packets. The detailed description is given in [10].

In this work, we further implement a new loadable module that can be loaded into the kernel of the backend servers. The module inserts itself between the network interface (NIC) driver and the TCP/IP stack. There are two purposes for the loadable module. First, the dispatcher will send the binding information to it, and then the module can change the outgoing packets so that they can go directly to the client without going through the dispatcher. As a result, the processing burden of the dispatcher can be greatly reduced, since the amount of data sent from the server to the client is significantly larger than the amount of data sent from the client to the server. The second purpose of such a design is for preventing the single-point-of-failure problem (which will be discussed in section 4).

C. Content-aware intelligence

With the above mechanism, the next question is how to build the request-discrimination intelligence into the dispatcher for making routing decision. To address this, we devise an internal data structure termed URL table to hold the content-related information, such as content size, type, priority, which nodes possess the content, etc. The dispatcher should consult the URL table to assign the incoming request to one of the backend servers. We argue that the URL table should model the hierarchical structure of the content in the Web site. Such an argument is based on an observation that people generally organize content using a directory-based hierarchical structure. That implies that the file in the same directory usually possesses the same properties. For example, the files underneath the /CGI-bin/ directory generally are CGI scripts for generating dynamic content.

Consequently, we implemented the URL table as a multi-level hash tree, in which each level corresponds to a level in the content tree and each node represent a file or directory. Basically, each item (file or directory) of content in a Web site should have a record corresponding to it in the URL table. However, to reduce the search time and the size of the table, our URL table supports a “wildcard” mechanism to specify a set of items that own the same properties. For example, if all items underneath the sub-directory “/html/” are all hosted in the same nodes and have the same content type, only the entry “/html/” exists in the URL table. If the
dispatcher intends to search the URL table to retrieve information pertaining to a URL “/html/misc.html”, it can get the information from the node “/html” in the table by just one level search. The URL table generally is self-generated, maintained, and managed by a management system [12] via parsing the content tree. The administrator also can configure the URL table if necessary.

III. REQUEST MIGRATION

In this section, we describe how an ongoing Web request can be smoothly migrated and recovered on another server node in the presence of server overload or failure. Our system requires a status-detection mechanism that could quickly identify the occurrence of server overload or failure. We do not discuss this issue in this paper, for which well-known solutions are available [e.g., 13,14,15]. We used the status-detection mechanisms implemented in our previous work [16], which could quickly detect the overloaded servers or the misbehaving servers.

We divide web requests into three types: requests for static content, requests for dynamic content, and requests for session-based services. We devise corresponding solution for each category. The dispatcher can identify the type of each request via consulting the URL table, and then migrate the requests of each category with the corresponding approach in case of server failure or overload.

A. Requests for static content

A majority of Web requests are to static objects, such as HTML files, images, and videos. We use the following mechanism to migrate such a request to another node. First, the dispatcher will select a new server (based on some load balancing mechanism), and select an idle pre-forked connection connected with the target server. Then the dispatcher re-binds the client-side connection to the newly selected server-side connection. After the new connection binding is determined, the dispatcher issues a range request on the new server-side connection to the selected server node. The range request is defined in the HTTP 1.1 [11] protocol, which allows a client to request portions of a resource. Using this property, we can enable a request to continue downloading a file from another node after the transfer was terminated in mid-stream.

From the TCP related information (i.e., ACK number, sequence number) recorded in the mapping table, the dispatcher can infer how many bytes the client has successfully received. As a result, the dispatcher could make a range request by including the Range header in it, specifying the desired ranges of bytes (generally starts from the last acknowledge number from the clients). Integrating with the technique of using pre-forked connection and seamlessly relaying packet between two TCP connections, we could smoothly recover a request on another node. It is noteworthy that the response of a range request has a unique HTTP header (e.g. carries the 206 status code), compared to the header of a usual response. Such a difference also needs to be translated by the module of backend servers.

B. Requests for dynamic content

Some web requests are for dynamic content (hereafter, dynamic request for short), for which responses are created on demand (e.g., CGI scripts, ASP), mostly based on client-provided arguments. At first, we tried to recover such dynamic request by replaying the request with the same argument to another node. However, we found this approach is problematic in some situations. The major problem is that some dynamic request is not “idempotent”, i.e., the result of two successive requests with the same arguments is different. The most common example is the dynamic Web pages constructed from the database. The two successive requests to the same page may be different due to the updates of the database. That means it is impossible to “seam” the results of the two requests by the range request approach described above. If we want to recover such dynamic request on another node, we should force the client to give up the data that it has received and then resubmit its request again. However, it will not be user-transparent and compatible with the existing browser.

As a result, we used the following approach to solve this problem. We made the dispatcher “store and then forward” the response of a dynamic request. In other words, the dispatcher will not relay the response to the client until it receives the complete result. Hence, if the server node fails in the middle of a dynamic request, the dispatcher will abort this connection, and then submit again the same request to another node. When it receives the complete result, it starts to reply to the client.

However, this approach has two potential problems. First, the dispatcher will suffer from the single-point-of-failure problem. We will discuss and solve this problem in section 4. Second, storing the server response in the dispatcher may have performance problem that negatively affects the throughput of the whole server system. We think the performance impact is not serious because the size of a dynamic web page is usually small [17,18]. However, to completely eliminate the performance concern, we made the dispatcher to function as reverse proxy (or termed as Web server accelerator [19]). That is, the dispatcher will cache the dynamic page so that the subsequent requests for the same dynamic page can access the content from the cache instead of repeatedly invoking a program to generate the same page. We implemented the algorithm proposed in [20] to manage the cached dynamic Web pages. As a result, the system not only can solve the failure recovery problem, but also significantly benefit from this approach in terms of performance. That is because the requests for dynamic content often slow down the web server considerably [21]. With such a design, the idempotent dynamic requests can be served directly from the memory cache, reducing the burden of the backend Web server.
C. Requests for session-based services

A so-called session consists of a number of user interactions. Here the user does not browse a number of independent statically or dynamically generated pages, but is guided through a session controlled by a server-side program (e.g., a CGI script) associated with some shared states. For example, such a state might contain the contents of an electronic "shopping cart" (a purchase list in a shopping mall site) or a list of results from a search request. These session-based services are generally based on so-called three-tier architecture, which is composed of the front-end client (e.g. browser), middle-tier Web server, and back-end database server. Basically, the front-end client provides user interface so that the user could use it to send the requests to the Web server for service access. The Web server implements the application/business logics. It processes the client's request, commits it to the database server, stores the resulting state, and returns the result to the client. Database servers manage data and transactional operation at the back end.

If a failure occurs at the web server in the middle of a session, the end-user typically cannot acquire any information what had actually happened, and whether the request was indeed committed or not. They may just wait until a timeout expires at the client side. Some users may re-send its requests, however these requests will not be answered since its state has been lost. Some users may re-submit all requests to retry the session, which will raise the risk of executing the transaction several times, having the user charged multiple times. These will annoy the users and severely tarnish the credibility of this Web site.

Recovering a session on another node is a more complicated problem. It requires knowledge of application-specific details such as when is the beginning of a session, internal state, intermediate parameter, when is the end of this session, and so on. We also need a mechanism to replicate the intermediate processing-state in order to ensure the fault-tolerance of the session itself. Now we tackle this problem by the following mechanism.

First of all, the web site manager should define a session for which fault resilience or higher performance is required. For example, the manager can define the action, "when a user adds the first item into a shopping cart on a specific web page," as a sign of the beginning of a session; and define the action, "when user clicks the check-out button," as the end of this session. The administrator could easily make such configurations via the GUI of our management system [12]. Such configuration information will be stored in URL table. As we described above, the dispatcher should consult the URL table to assign the incoming request to one of the web servers. When the dispatcher finds (here, we see again the benefit and necessity of content-aware routing mechanism) a request conveying the "start" action, it will "tag" this client and then direct all consequent requests from the client to one of the "twin servers", until it finds a request conveying the "end" action. The twin server is a logically couple of servers, which are composed of a primary server and a backup server. The primary server is a common Web server that is dedicated to provide transaction service; the backup server is responsible to backup the primary server.

The backup server has two IP addresses; one is its own address and the other is the address of the primary server. The IP address of the primary server is aliased on the network interface so that the local protocol stack will accept the corresponding incoming packets to that address. However, the backup server does not export the aliased address via ARP, since that would create IP-address conflict. In addition, one backup node can simultaneously serve as the backup of multiple servers. The twin server executes the following protocol (see figure 1).

When a client issues a request belonging to a session, the dispatcher will direct it to the primary server for processing. Because the backup server can receive all packets destined to the primary server by aliasing its address, the backup's HTTP daemon will also receive this request and then synchronize with the primary server by "silently" logging this request. As a result, the backup will keep the same "session processing state" with the primary. However, the backup's HTTP daemon does not deliver any packet or result unless the primary fails. This will guarantee that the client will only receive a single result.

When the backup has logged the request, it will send a "go for it" message to the primary for this request. Unless the primary receives such a message, the primary cannot commit this request to the database server. If the primary waits for the "go for it" message for a long time, it actively sends the backup a message to log this request and then waits for the message again. If it still cannot receive the "go for it" message, it suspects the backup has failed and then creates a new backup.

![Figure 1. Protocol for session-based requests](image-url)
When the primary receives the “go for it” message, it starts a traditional two-phase commit protocol with the database server, which will ensure the request agrees on the transaction semantic. When the database server replies the request, the primary sends the backup a message to log the outcome before it commits this request to the database server. After receiving the Ack message from the backup, the primary commits this request to the database server. When the database finishes this transaction, it replies an Ack message to the primary. At the same time, the backup could also receive this Ack message due to the aliased address. Then, it also sends an Ack message to tell the primary that it has logged the result of this transaction. After receiving both of the Ack messages (form backup and database), the primary sends the result page to the client. When both the primary and backup receive an Ack message from the client, they end the protocol and the backup releases the logged data if the whole session is over. When the primary server fails or is overloaded, the backup server can take over the job of the primary by activating the aliased address, and then the HTTP daemon populated on it start to deliver data instead of the primary server with the replicated state.

It is noteworthy that the dispatcher does not need to know whether the primary failed. The dispatcher always relays packets to the “representative” address (i.e. the aliased IP address). As long as the backup can take over the primary’s job and activate the aliased address, the service can continue even if the primary failed. Such a design could relieve the burden of dispatcher and enhance the overall performance.

We have extended the Apache [22] to realize the above protocol. We extended the Apache server to implement the protocol because of its open, reliability, efficiency and popularity in today's market. Apache follows the per-process-per-request model to handle the incoming requests. It pre-fork a set of processes and each process call accept() system call to accept new connection. Apache handles a request in a series of steps: (1) accepts a request, (2) parses its arguments for later use, (3) translates URL, (4) checks access authorization (5) determines MIME type of file requested, (6) processes the request, (7) sends response back to client (8) logs the request. We insert a decision-logic in step 6. If the process pertains to a primary server, it will send a log message to the backup server and wait for its reply. It will not submit the request to the database until it receives the backup’s reply. The log message is composed of source IP address and port number of the client, and the process’s ID. If the process pertains to a backup server, it will not “really” process this request but wait for the log message from the primary server. That means it will keep the same processing state with the primary server but skip the step 7.

IV. RELATED ISSUES

We found that the dispatcher may be a potential impediment to scalability due to the centralized design and software-based implementation, although the result of performance evaluation (see next section) shows that it scales well in a moderately sized server farm. We also noticed that the dispatcher represents a single-point-of-failure in our system, i.e., failure of the dispatcher would bring down the entire server system. To further improve scalability and fault tolerance, we can use multiple dispatchers to cooperate for distributing requests. In this configuration, the DNS approach [3] can be used to map different clients to different dispatchers.

We implemented a collection of daemon processes (based on the SwiFT toolkit [23,24]) that provides fault tolerance facilities on the group of dispatcher nodes, logically configured as a ring. Each dispatcher node runs the daemon process that monitors and backups its logical neighbor’s state. We refer interested readers to [23,24] for design and implementation details. All the dispatchers will participate in load sharing under normal operating conditions, i.e., no dispatcher is relegated to an idle hot standby status waiting for the failure of a primary dispatcher.

The dispatcher operates based on two important states: URL table and connection binding information. The URL table is a soft state that can be regenerated after the failure. In contrast, the connection binding information is a hard state that should be replicated in the backup node. Consequently, we made the primary dispatcher keeps a log of recent change of connection binding information, and periodically replicates the state change to its backup node to refresh the replicated table. If the primary fails, the backup can take over the primary’s job with the replicated state. However, in case of takeover, some state information for newly setup connections may be unavailable because the replicated state table has not yet been refreshed, or because the periodic refresh messages were lost. The kernel module in the backend server can cover this situation, since these modules also hold the connection binding information. If one server node does not receive packets from the dispatcher for a long time, it will broadcast a message to query the existence of the backup dispatcher, and then register its binding information to it.

In this paper, we concentrate on the design and implementation of our request migration mechanism. This paper clearly demonstrates that our mechanism offers a powerful solution to support fault-resilience for Web services. However, due to space limitation, we pay little attention to the policy such as algorithms for deciding which server to forward the failed request to, or algorithms for deciding which requests should be migrated in case of server overloaded. The readers are referred to [25] for further information. That paper describes an algorithm that uses this mechanism to guarantees the important services against the loss due to server overload.

V. SYSTEM EVALUATION

This section presents the results of performance evaluation on the proposed mechanism. Due to the space limitation, we only report the major data.
A. Methodology

We constructed the following test environment in the laboratory for performance evaluation. We used a Pentium-2 (350 MHz CPU with 128 MB memory) machine running Linux (version 2.2.12) to serve as a dispatcher. The server cluster consists of the following machines: four Pentium Pro (200 MHz CPU with 64MB memory) machines and six Pentium-2 (300 MHz CPU with 128 MB memory). Each server node runs Linux with Apache (version 1.2.4). We connected all these machines directly by a 3Com 3300 switch using 100Mbps full-duplex network connections.

We used 24 Pentium-2 (350 MHz CPU with 128 MB memory) machines to run the WebStone [26] benchmark for generating a synthetic workload to evaluate the proposed system. Each machine runs four WebStone client programs that emit a stream of Web requests, and measure the system response. The generated loads are varied in experiments by varying the number of WebStone clients.

The stream of requests is called the workload. We created a workload that models the workload characterization (e.g., file size, request distribution, file access frequency, etc.) of representative Web servers [17,18]. The workload was configured with approximately 6000 unique files of which the total size is about 116MB. Details of the workload are given in appendix A. We also inserted the session model into the workload so that session-based service could be investigated. We implemented a session generator in the WebStone client to issue session-based requests.

The session generator is driven by a state machine that models the request stream of a session issued by a user. The so-called states include Login, Browse, Search, Select, Add to the Shopping Cart, delete an item from the Shopping Cart, View account, Pay, and Logout, etc. The session generator issues a stream of requests based on the probability of state transfer, which mimics how users change state as they logon to the site, going from browsing to searching, selecting items, adding them to their shopping carts and paying. The idea behind our state machine is very similar to the customer behavior model graph (CBMG) [27] that captures how users navigate through an e-commerce site. Consequently, we omitted the detailed description of the operation of the session generator for the sake of brevity and space limitations. In the following experiments, fifteen percent of the total requests are session-based requests.

We repeated each test for a number of iterations in each experiment. Each trial included a warm-up stage, not timed, so initial activity did not influence the reported results. The results presented are mean values of 20 trials for each data point. The standard deviations for all results are generally less than 5 percent of mean.

B. Request-migration time

In the first experiment, we want to quantify the performance degradation when request migration occurs. We generated a heavy load (128 clients) with the workload defined in appendix A to evaluate the server system. The generated load was able to push the server into a prolonged overload state. We implemented a fault-injection program executing on each server nodes, which will shutdown and restart the system or HTTP daemon to simulate failures and repairs. The average Mean Time between Failures (MTBF) is about 5 minutes, and Mean Time to Repair (MTTR) is 30 seconds.

The WebStone clients measured the average response time for each HTTP request. Response time is defined as the interval between a request submission and the end of the corresponding response from the system. We identified the requests affected by the failed server and measured their response time. Compared with the average response time of normal requests (i.e., requests that were not affected by the failures), we could quantify how much latency will increase due to request migration in the presence of server failures. We discuss the additional latency of request for static content, dynamic content, and session-based service, respectively.

1) Request for static content

The additional latency of requests for the static content is given in Table 1. The values in the “baseline” are the response time of normal requests that were not affected by the failures. The values in the second and sixth rows are response time of requests affected by server failures. The result shows that the requests only experience relatively small performance degradation when request migration occurs.

It is noteworthy that the additional latency of the small or moderate request (below 256k) is independent of request size. In contrast, the fail-over time of the large requests (requests for 1024k) is far higher than the other requests. We found that this was caused by the caching effect of the backend server. Our workload models the typical content popularity of a web site, which often follows a Zipf distribution [17]. The large files are usually unpopular so that their access frequency is very low, with the result that such large files usually cannot be found in the cache memory. As a result, the web server has to obtain the content from disk and then retrieve the needed portion to continue the interrupted service, resulting in a higher additional latency.

<table>
<thead>
<tr>
<th>Request size (Kb)</th>
<th>4K</th>
<th>8K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failed request (ms)</td>
<td>885.23</td>
<td>924.38</td>
<td>1080.24</td>
</tr>
<tr>
<td>Baseline (ms)</td>
<td>22.24</td>
<td>32.12</td>
<td>151.38</td>
</tr>
<tr>
<td>Additional Latency</td>
<td>862.99 ms</td>
<td>892.26 ms</td>
<td>908.86 ms</td>
</tr>
</tbody>
</table>

We think that the above measured latency cannot precisely represent the “request-migration time”, because some of the time should be attributed to the failure detection latency. To precisely quantify the required time of request migration, we instrumented the Linux kernel of the dispatcher and Apache server of back-end nodes. We also ran the
The timeline in Figure 2 depicts the request-recovery steps and related metrics in the presence of failures.

\[ \text{Figure 2. Timeline and metrics} \]

\[ T_d \] is the failure detection latency. As we described above, we think \( T_d \) is charged to the measured latency in the previous measurement. \( T_r \) is the time that dispatcher spends on invoking request recovery and forging the partial HTTP request. Here, we see the benefit of pre-forking and re-using TCP connection. The request can be immediately sent to the new server without waiting for creation of a new connection. \( T_c \) is the network latency that the request takes to reach the server. \( T_{\text{par}} \) is the queuing delay that the HTTP request spends at the server pool process, and the time that the server spends on reading and parsing the HTTP request. \( T_{\text{proc}} \) is the request-processing time that the server spends on moving the requested content from disk (or cache) to the network. \( T_{\text{net}} \) is the network latency that the response packet takes to reach the server.

The result of instrumentation is given in Table 2. The maximum request-migration time is below 300 ms, which is well below the time that could cause user’s TCP connections to time-out. Based on these results, we could conclude that our mechanism can enable a static request to be smoothly migrated to another working node with relatively small performance degradation.

\[ \text{Table 2. Result of instrumentation} \]

<table>
<thead>
<tr>
<th>Request size</th>
<th>( T_r ) (ms)</th>
<th>( T_c ) (ms)</th>
<th>( T_{\text{par}} ) (ms)</th>
<th>( T_{\text{proc}} ) (ms)</th>
<th>( T_{\text{net}} ) (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k</td>
<td>3.23</td>
<td>2.58</td>
<td>8.38</td>
<td>9.53</td>
<td>2.25</td>
</tr>
<tr>
<td>8k</td>
<td>3.12</td>
<td>2.35</td>
<td>8.52</td>
<td>8.89</td>
<td>3.01</td>
</tr>
<tr>
<td>32k</td>
<td>4.18</td>
<td>2.21</td>
<td>9.86</td>
<td>10.23</td>
<td>2.05</td>
</tr>
<tr>
<td>64k</td>
<td>3.89</td>
<td>2.54</td>
<td>8.25</td>
<td>10.56</td>
<td>2.16</td>
</tr>
<tr>
<td>256k</td>
<td>3.56</td>
<td>2.39</td>
<td>7.89</td>
<td>42.56</td>
<td>2.38</td>
</tr>
<tr>
<td>1024k</td>
<td>3.09</td>
<td>2.64</td>
<td>8.69</td>
<td>225.23</td>
<td>2.53</td>
</tr>
</tbody>
</table>

2) Request for dynamic content

The dynamic content that we created in the workload can be divided into three categories: light, moderate, and heavy. We give a brief description, average response time, and average additional latency of each category in Table 3.

The result shows that the additional latency of recovering a request for dynamic content is very acceptable. This low latency could be attributed to the design that the dispatcher stores and then forwards the dynamic page. Most of the requests are replied to by the dispatcher from the memory cache. These requests were not affected by the server failures. The worst case was when the dynamic page could not be served from the cache, thereby the dispatcher should invoke the backend server to generate the content, at the same time, the corresponding server fails before it can send the complete response to the dispatcher. In this case, the dispatcher should abort the ongoing connection, send the same request to another working node, and then serve the client. The moderate and large requests have a higher probability to fail in such worst case scenario, thereby causing the higher fail-over time.

We recognize that the true problem may not be the low additional latency in case of recovery, but if the user perceived performance suffer from the cost associated with the design of caching dynamic content. We will investigate this concern in the subsection C.

3) Session-based requests

The average additional latency caused by recovering the session-based requests is about 1.25 sec. Similarly, we think the failure detection latency contributed to the significant portion of the measured fail-over time. In fact, when the backup server discovers the failure of the primary, it could quickly take over its job and immediately reply to the client.

C. Overhead and scalability

The major concern of our mechanism is how much cost is associated with it when no failure occurs, and if the regular system’s performance will suffer from the additional overhead. To quantify the additional overhead, we measured and compared the response time of a Web request in our system, with those in a baseline system without the proposed mechanism. The baseline system was the server clusters (as the configuration described above) front-ended by a TCP connection router that is the implementation in our previous work [7]. The TCP connection router dispatches requests based on layer-4 routing, which is commonly used in many server-load-balancer products. The overhead introduced by the proposed mechanism is defined to be the difference in response time between the two situations. We used the same client system and workload to test the two configurations. Similarly, we discuss each type of request respectively. The results of requests for static content are given in Table 4. The results show that the additional latency caused by the proposed mechanism is insignificant.

\[ \text{Table 3. Additional Latency (dynamic content)} \]

<table>
<thead>
<tr>
<th>Type</th>
<th>Response Time</th>
<th>Additional Latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>0.82 sec</td>
<td>14.64 ms</td>
<td>Content generated by ASP</td>
</tr>
<tr>
<td>Moderate</td>
<td>2.64 sec</td>
<td>1.42 sec</td>
<td>Content generated by CGI script</td>
</tr>
<tr>
<td>Heavy</td>
<td>6.17 sec</td>
<td>2.34 sec</td>
<td>Complex database query by CGIdscript</td>
</tr>
</tbody>
</table>
### Table 4. Overhead (Static content)

<table>
<thead>
<tr>
<th>Request size (Kb)</th>
<th>4K</th>
<th>8K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Our system (ms)</strong></td>
<td>27.19</td>
<td>36.07</td>
<td>174.58</td>
</tr>
<tr>
<td><strong>Baseline (ms)</strong></td>
<td>23.58</td>
<td>32.25</td>
<td>170.24</td>
</tr>
<tr>
<td><strong>Overhead (ms)</strong></td>
<td>3.61</td>
<td>3.82</td>
<td>4.34</td>
</tr>
<tr>
<td>Request size (Kb)</td>
<td>64K</td>
<td>256K</td>
<td>1024K</td>
</tr>
<tr>
<td><strong>Our system (ms)</strong></td>
<td>312.9</td>
<td>1151.04</td>
<td>4824.1</td>
</tr>
<tr>
<td><strong>Baseline (ms)</strong></td>
<td>308.39</td>
<td>1145.62</td>
<td>4815.17</td>
</tr>
<tr>
<td><strong>Overhead (ms)</strong></td>
<td>4.51</td>
<td>5.42</td>
<td>8.93</td>
</tr>
</tbody>
</table>

The additional overhead mostly comes from the content-aware mechanisms. To perform request discrimination, the dispatcher should look into the HTTP header of each request. In our current design, the dispatcher only looks up the "first line" (i.e., request line [11]) to make routing decision. That does not cause too much overhead. At the period of peak throughput, the CPU utilization of the dispatcher is 62%, and the consumed memory of our system is slightly larger (only 2.3 Mbytes) than that of the baseline system. This means that our mechanism is not a performance bottleneck. However, our experience showed that the system performance would be severely degraded if we look deeper in the HTTP header for retrieving more information (e.g., Host, Cookie) to perform request routing. This observation motivated us to devise a new mechanism termed “URL Formalization” [12] to address this performance concern.

In addition, for each incoming HTTP request, the URL table must be consulted to make the routing decision. As a result, the process of locating the entry relevant to each incoming request can become a significant performance factor. As we described above, we used a sophisticated data structure, efficient hash function, and cache mechanism to diminish the overhead. We are aware that the overhead associated with consulting the URL table is dependent on the size of the table and therefore on the amount of data stored in a Web site. As a result, we measured this overhead from our Web site running the proposed system. Our Web site contains about 11700 Web objects. In such scale, the memory consumed by the URL table is about 540k bytes. During the peak load, the average lookup time is about 3.62 µsec, which is still insignificant. Please notice that the overhead is charged per request not per packet.

The results regarding the requests for dynamic content are given in table 5. We do not see any significant overhead introduced by the “store and then forward” mechanism. We performed the same experiment several times, and got a similar result. In the case of the heavy requests, our system even performed better than the baseline system. This is because such requests are for searching the database with some keywords. The used arguments of each request are the same, resulting in the same response content, thus they can be served directly from the cache of the dispatcher.

### Table 5. Overhead (dynamic content)

<table>
<thead>
<tr>
<th>Type</th>
<th>Baseline</th>
<th>Our system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>0.84 sec</td>
<td>0.85 sec</td>
</tr>
<tr>
<td>Moderate</td>
<td>3.12 sec</td>
<td>3.13 sec</td>
</tr>
<tr>
<td>Heavy</td>
<td>5.43 sec</td>
<td>1.25 sec</td>
</tr>
</tbody>
</table>

In terms of session-based requests, our protocol introduces an overhead of about 7% over the baseline system that does not offer any guarantee. The additional latency mostly comes from the need to wait for the backup node to store the processing state. With the IP aliasing technique, backup and primary usually receive the request simultaneously, thereby reducing the logging latency. Notice that the experiment was measured over a local area network, where high-speed connections are the norm, resulting in short observed response time and then large relative overhead. The overhead would be insignificant when compared with the typical latency over wide-area networks. The low cost is because our fault resilience mechanism does not need to record much state for each request, and the request migration could be effectively achieved by re-bind the user-side connection to another pre-forked connection.

### D. A Long-running test

This section presents the performance data of a long-running test that lasted for about 3 days. We used the WebStone with the same workload to perform this experiment. The benchmark clients constantly generated high load (1200 requests/sec in this experiment) to the server system, and measure the response time and throughput (request per second) on the interval of per second. The fault-injection programs simulated the failures and repairs in a pre-defined probability distribution. The average Mean Time between Failures (MTBF) is 1 hour, which is far higher than the value in the real world [29,30].

The most noteworthy result of this experiment is that the total error rate of this experiment is zero. That is, almost no request fails despite some nodes of the server system fail. Figure 3 depicts an annotated timeline of the events that occur during the failure of 1 and 3 nodes respectively. In the latter case, we intentionally made three server nodes fail simultaneously. The throughput drops dramatically as some nodes fail. When one server node fails, the outstanding requests on the failed node were smoothly migrated to another working node. When three nodes fail, the dispatcher discovered that the system was overloaded and then recruited two spare nodes to share the load. After 7 seconds, the system returned to stabilization. This experiment also can serve as a proof that the request migration mechanism can relief the server overloads.

![Figure 3. A Long-running test](image-url)

In terms of session-based requests, our protocol introduces an overhead of about 7% over the baseline system that does not offer any guarantee. The additional latency mostly comes from the need to wait for the backup node to store the processing state. With the IP aliasing technique, backup and primary usually receive the request simultaneously, thereby reducing the logging latency. Notice that the experiment was measured over a local area network, where high-speed connections are the norm, resulting in short observed response time and then large relative overhead. The overhead would be insignificant when compared with the typical latency over wide-area networks. The low cost is because our fault resilience mechanism does not need to record much state for each request, and the request migration could be effectively achieved by re-bind the user-side connection to another pre-forked connection.

### D. A Long-running test

This section presents the performance data of a long-running test that lasted for about 3 days. We used the WebStone with the same workload to perform this experiment. The benchmark clients constantly generated high load (1200 requests/sec in this experiment) to the server system, and measure the response time and throughput (request per second) on the interval of per second. The fault-injection programs simulated the failures and repairs in a pre-defined probability distribution. The average Mean Time between Failures (MTBF) is 1 hour, which is far higher than the value in the real world [29,30].

The most noteworthy result of this experiment is that the total error rate of this experiment is zero. That is, almost no request fails despite some nodes of the server system fail. Figure 3 depicts an annotated timeline of the events that occur during the failure of 1 and 3 nodes respectively. In the latter case, we intentionally made three server nodes fail simultaneously. The throughput drops dramatically as some nodes fail. When one server node fails, the outstanding requests on the failed node were smoothly migrated to another working node. When three nodes fail, the dispatcher discovered that the system was overloaded and then recruited two spare nodes to share the load. After 7 seconds, the system returned to stabilization. This experiment also can serve as a proof that the request migration mechanism can relief the server overloads.

In terms of session-based requests, our protocol introduces an overhead of about 7% over the baseline system that does not offer any guarantee. The additional latency mostly comes from the need to wait for the backup node to store the processing state. With the IP aliasing technique, backup and primary usually receive the request simultaneously, thereby reducing the logging latency. Notice that the experiment was measured over a local area network, where high-speed connections are the norm, resulting in short observed response time and then large relative overhead. The overhead would be insignificant when compared with the typical latency over wide-area networks. The low cost is because our fault resilience mechanism does not need to record much state for each request, and the request migration could be effectively achieved by re-bind the user-side connection to another pre-forked connection.
VI. RELATED WORK

Most of the prior work on the web server cluster has concentrated on request distribution among a group of server nodes. We have listed and discussed some of them in section 1. This is also an active area that a lot of vendors are working on, and there are a number of commercial products (e.g., [31,32]) that have been announced. However, server systems clustered by these schemes or products merely provide high availability by the redundant nature. They do not address the fault resilience problem.

The idea of content-aware routing is not new. In addition to us, a number of research works [33,34,35] or commercial products [36-41] also proposed a similar idea. The contribution of this work is that we integrate the content-aware routing, connection pre-forking and re-using, seamlessly relaying packets between two TCP connections, and the fault recovery mechanism to effectively provide the capability of fault resilience for the Web services. We believe that the mechanisms proposed in this paper could be easily incorporated in those commercial products.

Ingham et al. [42] surveyed some existing approaches for constructing a highly dependable Web server. They also pointed out the importance of providing transactional integrity and found that it is not addressed in the existing system. HAWA [43] addressed the service availability problem in client side by an applet-based approach. This is a very interesting approach and has the advantage of low overhead. However, they do not address the server-side fault tolerance problem and deal with the transaction-based service. Singhai et al. [44] and Chawathe et al. [45] proposed a framework for building a highly available Internet service. These systems indeed could provide higher availability, however, they are not fault-tolerant and highly reliable. None of these approaches or systems addresses the issue of request migration in presence of server failure and smooth provision of service while migration occurs. Frolund et al. [46] have done a great and important work in providing fault tolerance for E-transaction. We found that our twin server protocol share the same principle with part of their protocol. Consequently, their theory can serve a proof to our protocol.

VII. CONCLUSION

We have described our design, implementation, and performance data of a novel mechanism that enables request migration in the Web server cluster. We seek to ensure that no Web request is lost as a result of a server failure or overload. The new mechanism is a powerful solution to providing fault tolerance and dynamic load-distribution for Web services. It is well known that providing fault tolerance often comes at a considerable cost and associated overhead. In this paper, we have showed that high service reliability can be effectively achieved by clever design and careful implementation. The results of performance evaluation demonstrate that our approach is efficient and with low cost.

We believe that our approach takes an important step toward providing a highly dependable Web service.

ACKNOWLEDGEMENT

This work was supported by the National Science Council, R.O.C., under contract no. NSC 89-2622-E-110-003.

REFERENCE

[38] F5Labs. BigIP. http://www.f5.com/

Appendix A.

<table>
<thead>
<tr>
<th></th>
<th>Number of Files</th>
<th>Average File Size (bytes)</th>
<th>Request Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP</td>
<td>37</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>CGI</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>CLASS 1 (gif)</td>
<td>301</td>
<td>223</td>
<td>12</td>
</tr>
<tr>
<td>CLASS 2 (gif)</td>
<td>200</td>
<td>735</td>
<td>6</td>
</tr>
<tr>
<td>CLASS 3 (gif)</td>
<td>361</td>
<td>1522</td>
<td>8</td>
</tr>
<tr>
<td>CLASS 4 (jpg)</td>
<td>665</td>
<td>2895</td>
<td>15</td>
</tr>
<tr>
<td>CLASS 5 (htm)</td>
<td>1865</td>
<td>6040</td>
<td>12</td>
</tr>
<tr>
<td>CLASS 6 (htm)</td>
<td>1705</td>
<td>11426</td>
<td>14</td>
</tr>
<tr>
<td>CLASS 7 (htm)</td>
<td>721</td>
<td>22132</td>
<td>8</td>
</tr>
<tr>
<td>CLASS 8 (htm)</td>
<td>265</td>
<td>41518</td>
<td>1</td>
</tr>
<tr>
<td>CLASS 9 (exe)</td>
<td>53</td>
<td>529k</td>
<td>2</td>
</tr>
<tr>
<td>CLASS 10(Video)</td>
<td>27</td>
<td>1024k</td>
<td>1</td>
</tr>
</tbody>
</table>