System Support for Scalable, Reliable and Highly Manageable Web Hosting Service

Mon-Yen Luo and Chu-Sing Yang
Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C

Abstract

This paper presents the architecture and some key mechanisms of an integrated framework for providing a reliable and highly manageable Web hosting service on a scalable server cluster. We devise a novel idea termed "*URL Formalization*" and a corresponding data structure, which provide a scalable solution to the request distribution in the system. We exploit the advantages of Java to implement a management system to providing a highly manageable system. Our system supports a higher level of services reliability than other server cluster systems. The result of performance evaluation on the proposed system shows that the system is low-cost and effective.

1. Introduction

With the popularity of the Internet and the World Wide Web, the desire for using the Web to serve business transactions is increasing at an amazing rate. A successful Web site has become increasingly essential to the business community. However, constructing a successful Web site must cope with many challenging problems. First, a web site must be able to service thousands of simultaneous client requests and scale to rapidly growing user population. Furthermore, rapid response and 24-by-7 availability are mandatory requirements for a Web site as it competes for offering users the best "surfing" experience. As a result, Web hosting service is soaring as companies turn to service providers to handle these challenges, avoid infrastructure costs, and deal with staffing shortages. The Web hosting service providers offer system resources (e.g., bandwidth to the Internet, disks, processors, memory, etc.) to store and provide Web access to documents from individuals, institutions, and companies who lack the resource or the expertise to maintain a Web site.

Web server cluster [1] is a popular approach used in a shared Web hosting infrastructure as a way to create scalable and highly available solutions. However, hosting a variety of contents (i.e., documents, Web pages, resources, etc., generally called content in the Internet parlance) from different customers on such a distributed server system faces new design and management problems and requires new solutions. This paper describes the research work we are pursuing for constructing an integrated framework to address the challenges faced by hosting Web content on a server cluster environment. Our system has a variety of design goals, however, this paper focus on addressing the following two major problems:

- **Content management**
  How to place and manage hosting content in a clustered server will become a very important but challenging problem. You may imagine how tedious and difficult if you have 500 different sites with a variety of content types, and you try to place and manage these contents on a server cluster with 60 nodes. Thus, developing a management system to automate management operations and provide a logical view of a monolithic system is extremely important. In particular, such clustered servers tend to be more heterogeneous because they generally grow incrementally as needed. This is an important advantage of clustered servers so that they can scale gracefully with offered load, preserving previous investments in hardware and software. Unfortunately, this advantage will come at the cost of greatly increased management complexity.
- **Service reliability**
  The existing clustered servers merely provide high availability, but offer no guarantees about fault resilience for the service. If one server node fails, most of the existing clustered systems can mask this failure and then reconfigure the system. However, any requests in progress on the failed server will fail. This will be unacceptable to the customers who conduct E-transaction services on the Web. In addition to detecting and masking the failed nodes, an ideal fault-tolerant server should enable the ongoing requests on the failed node to be smoothly migrated and then recovered on another working node. However, this is a challenging problem, and none of the existing clustering schemes could efficiently support this capability.

Basically, our system consists of a request distributing mechanism and a Java-based management system. The distributing mechanism presents a single entry point to the external world, and manages the allocation of incoming requests to server nodes. To effectively support request distribution in the shared hosting environment, we devised a novel idea termed "*URL Formalization*" and a corresponding data structure. In addition, we also augment the distributing mechanism with a novel mechanism termed *request migration*, which transparently enables request migration and recovery in the Web server cluster in the presence of server overload or failure. The Java-based management system could relieve the administrator's burden on managing the complex system.

The rest of this paper is organized as follows. In section 2, we will describe our distributing mechanism and how it can provide a scalable solution to the request distribution in our system. The section 3 presents how we exploit several advantages of Java to implement a management system to provide a highly manageable system. In section 4, we describe how a higher reliability can be achieved by our system. We present the result of performance evaluation on the prototype system in section 5. We discuss the advantages and possible arguments of our system in section 6, and then draw conclusions in section 7.

2. Distributing Mechanism

Given a clustered server, some distributing mechanism is needed to dispatch and route the incoming requests to the server best suited to respond. In this section, we describe the design and implementation details of our distributing mechanism.

2.1 Content-Aware Routing

Over the past few years, numerous distributing mechanisms have been proposed. These schemes can be classified into the following categories: client-side approach [2,3], DNS based approach [4,5], TCP connection routing [6,7], HTTP redirection [8], and content-based routing [9,10,11]. We think that the content-based routing mechanism is the best choice to the Web hosting environment, because the other schemes only can perform request routing based on some simple criteria (e.g., number of ongoing connections). As Web services became more sophisticated, such simple routing schemes are no longer sufficient. New services such as online retailing began to use multiple tiers of servers for content and transaction processing, posing more requirements and challenges to the routing mechanisms. The content-aware routing mechanism can offer many potential benefits [11], such as sophisticated load balancing, QoS support, session integrity, flexibility in content deployment, etc.

The distributing mechanism of the proposed system is based on our previous work [11] on implementing a content-based routing mechanism. We briefly describe the operation of the content-based routing mechanism as follows; the detailed description is given in [11]. The dispatcher node that executes the routing mechanism pre-forks a number of persistent connections [12,13] to the back-end nodes, and then allocates system resources by dispatching client requests on these trunks. When a client tries to retrieve a 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 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 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 client and the server can transparently receive and recognize these packets.

In this paper, we further point out that such a design can enable a new capability: *request migration.* The request can be migrated to another node in the presence of server overload or server failure, which can control the resource allocation in a fine granularity and enhance service reliability, respectively. In this paper, we focus on the aspect in terms of service reliability. To support high reliability, we think the server cluster should be augmented to include two important capabilities: checkpointing and failover. That is, some intermediate state of user requests should be logged periodically by the checkpointing mechanism. If one server fails, the failover mechanism should enable the ongoing requests on the failed node to be continued processing with a valid intermediate state in another working node. Although the two techniques has been investigated and are well known in the research area of fault tolerance, implementing these techniques in the distributed web server still poses many new challenges.

First, the cost is very expensive if we log every incoming request for checkpointing. In a Web hosting

system, not all content is equally important to the client and the service provider. Some of the hosting contents cannot tolerate service disruptions because of their importance or cost. With the content-aware routing capability, the distributing mechanism can differentiate the important requests (e.g., requests for mission-critical services, or requests for content owned by important customers) from regular Web surfing requests. Second, how to recover a Web request of a failed server to continue execution in another working node is a challenging problem. In particular, such recovery mechanism should be user-transparent and smooth. In the section 4, we will describe our design that provides an elegant solution to this problem.

2.2 Content-Aware Intelligence

With the above mechanism, the next question is how to build the content-aware intelligence into the dispatcher for making routing decision. To address this, we should have answers to the following questions:

- What kind of information do we need?
- How can we put the related information into the dispatcher?
- How can the dispatcher perfom content-aware routing based on these information?

To the first two questions, we devised an internal data structure termed *URL table* to hold the content-related information that enables the dispatcher to make intelligent routing decisions. The possible information includes content size, type, priority, which nodes possess the content, etc. We argue that the URL table should model the hierarchical structure of the Web content. Such an argument is based on the observation that people generally organize content using a directory-based hierarchical structure. The files in the same directory usually possess 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 uses 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 the information about a URL "/html/misc.html", it can get the information from the record "/html" in the table by just one level search. Otherwise, we also implemented a mechanism to cache recently accessed entries, which is a proven technique for searching speedup. The URL table generally is self-generated, maintained, and managed by the management system (see next section) via parsing the content tree. The administrator also can configure the URL table if necessary.

Our previous experience [11] has taught us that to address the third question is a more thorny challenge. To perform content-based routing, the dispatcher should look into the HTTP header of each request. However, the HTTP header is composed of variable-length strings. Therefore, performing content-based routing implies that some kind of string search and matching algorithm is required. It is well known that such operations are time consuming. Our experience showed that the system performance would be severely degraded if we implement some string parsing functions in the dispatcher. Some vendors also made a similar observation [14], which indicated that you will loss 7/8ths of your Web switch's performance if you turn on its URL parsing function.

The above problem will be more serious in the shared Web hosting environment. We consider the following URL: *http://www.foo.com/sports/football/* as an example for explanation. This URL identifies that the content in the directory "/sports/football/" on the host "www.foo.com" can be accessed via HTTP protocol. A client wishing to retrieve this resource should create a TCP connection to the server and send a HTTP request including a request line [13] in its header:

Get /sports/football/ HTTP 1.1

, followed by the remainder of the request. The Host name of the URL will be transmitted in a Host header field of the entity-header fields [13], followed by the request line. Because all Web sites in the shared hosting environment are publicized by the same IP address to the external world, the host field is required to identify which Web site the requests is for. This implies that the dispatcher needs to look deeper in the HTTP header (not just the request line) to find the host field. As the HTTP header is composed of variable-length strings, parsing the header to retrieve such information will be a considerable burden.

To solve this problem, we devised a novel mechanism termed *URL Formalization*. Our approach is to make every directory and file of the Web content have a formalized expression. In our system, all Web objects originally reside on a reliable "home server", which is also the place for the customers to upload their content. The document stored on the home server also serves a permanent copy for consistency and robustness. Before these web objects are placed to the server system, a

program will convert the original name of every directory and file into a fixed length and formatted name. Then, the program will parse all html files and script files that generate dynamic content, and modify the embedded hyperlinks to conform to the new name. In addition, the new path name of each link will be converted to a composition of the original path names under its domain name. Finally, the content is placed to the server nodes as the converted name. But, they also have the original name as an aliasing name.

For example, if an embedded link points to the above URL, the link should be converted to "/preamble/5967/1019/2048/". The name "www.foo.com", "sports", and "football" are converted to a formalized name 5967, 1019, and 2048 respectively. The preamble is a "magic number", which is designed to indicate that the following is a formalized URL. This also implies that the name of the first level directory of each server node is "/preamble", and the hosted content is placed under the directory. The design of the preamble number is important, because we should enable the dispatcher to know whether the URL of a request is in normal form or formalized form. The operations of parsing and reconstructing the HTML files and the script files are pre-computed offline. Thus, these operations do not impose any performance penalty on regular operations of the server system.

In the URL formalization scheme, the request line of the HTTP header in the above example will be:

Get /preamble/5967/1019/2048/ HTTP 1.1

The major advantage of such a design is to convert user-friendly names to routing-friendly names. In other words, our fixed-length and formalized names are easier for dispatcher to process. We even can implement the routing function in hardware for performance boosting. In addition, the artful design of placing the host name in the first level of the path name can further speed up the routing decision. The dispatcher can quickly identify that the incoming request is for which Web site, rather than parse the entire HTTP header to find out the host field. Combined with the well-designed URL table, the dispatcher can quickly retrieve related information to make routing decision.

The design of the URL formalization is based on the following observations. Generally, the reason for using the variable-length string to name a file or directory is just because it is mnemonic, thereby making it easier for humans to remember. However, in most cases an HTTP request is issued when the browser follows a link: either explicitly, when the user clicks on an anchor, or implicitly, via an embedded image or object. That is, most URLs are invisible to the users, i.e., they do not care about what name it has. Consequently, we can convert the original name to a formalized form in the manner of user transparency. However, in the relatively infrequent case where users occasionally load Web pages by typing a URL directly. That is why the magic number as a preamble is necessary, so that the dispatcher can distinguish the regular URL from the formalized URL.

3. Java-based Management System

In this section, we first consider issues that arise in designing a management system for hosting service in server cluster environment. Following this, we describe our technical design and implementation.

3.1 Analysis

Placing and managing hosting content in a server cluster environment is not a trivial job; it needs to take some important factors into consideration. First, the hosting content might be as varied as static Web pages, dynamic content (e.g., generated by a CGI script), or multimedia data such as streaming audio or video. Different contents have different requirements in terms of system resources. For example, requests for executing CGI scripts normally require much more computing resources than static file retrieval requests [15]. It is clear that some nodes with slow processors are not suitable for providing CPU-intensive dynamic content. Second, not all content is equally important to the service provider. The variety of content from different customers means that the expectations and requirements on the quality of service differ. In addition, the amount of money each customer is willing to spend and can afford to pay differs. The administrator needs to be able to place different content on different nodes for meeting their performance requirements, or to exert explicit control over resource allocation according to the variety of service agreement.

This problem motivated us to design a management system to ease the administrative operations and provide a logical view of a monolithic system to the system manager. We intended to construct a group of daemons running on each node, and enable them to cooperate to perform a variety of management functions. However, many problems arose when we tried to implement such an idea. The first major problem is platform heterogeneity, which arises from that we hope the proposed system is flexible enough that each server node can use any kind of hardware, operating system, and Web server software. This implies that these daemons must be capable of running on different platforms. The second considerable problem is *function heterogeneity,* which is because each server node might need different management functions due to hosting different content. The third

problem is in terms of extensibility (or versioning and distribution problem). That is, the functionality of this management system cannot be extended or customized without rewriting, recompilation, re-installation, and re-instantiation of all existing daemons. In particular, each service provider may have it's own unique requirement.

3.2 Management Framework

For tackling the above problems, we decided to construct the management system using Java [16]. The management system is an extension of our previous work [17], which is an extensible framework to address the administration problem of a Web server cluster. Our management system is composed of the following four key components: *controller*, *broker*, *managelet* (composed of two words: manage and –let, it means a piece of code implementing a specific management function), and *remote console* (see figure 1).

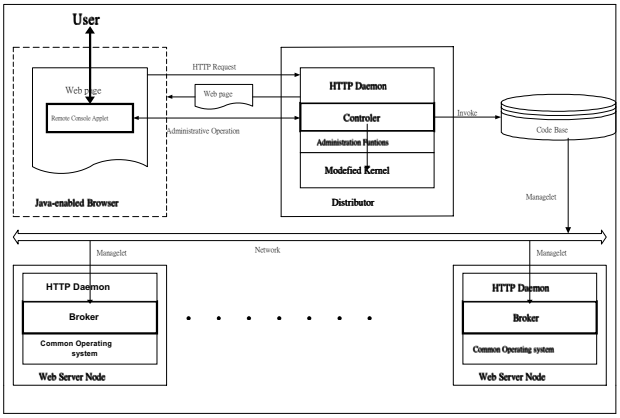

Figure 1. Overview of the Management system

The broker runs on each Web server node to perform the management functions, and monitor the status of the managed node. The broker can load Java code from the network and provides an environment for executing it. To achieve this, we implemented a customized Class Loader and Security Manager [18] as the skeleton of the broker. We also implemented a monitor thread in the broker daemon to collect the load information of the host on which it is located. The collected load information will be summarized and then sent to the dispatcher node. Each management function is implemented as a corresponding managelet, which is in the form of Java class. These managelets are stored in a reliable storage. One special daemon called *controller* will be responsible for receiving requests from the administrator, and then invokes brokers to perform the delegated tasks by dispatching the corresponding managelet to be executed on them. The remote console is a Java applet, which can be run on any Java-enabled Web browser. The administrator can download the remote console and interact with it to perform management operations.

There are two main advantages of such a design. First, implementing the daemon in Java can relieve the concerns related to heterogeneity of the target platforms. As a result, these daemons can be capable of executing on a variety of hardware architectures and operating systems. The second advantage is derived from the notion of downloaded executable content (data that contain programs that are executed upon receipt), which is a powerful feature of Java. Using this mobile code technology, we can deploy just a simple local daemon (broker) at each node, but support a variety of management functions via downloading managelets. As a result, the system can be tailored or extended to the different requirements of different system types and installations, without requiring significant redesign and coding. In the following sections, we will describe how the management system addresses some management problems of hosting content on a server cluster.

3.3 Configuration Management

We provided several administration functions on the dispatcher for the administrator to configure the server system. Examples include join/remove a server node into/from the system, read the related statistical data, etc. We defined the control interface as the boundary between these native configuration functions pertained to the dispatcher and the rest of the management system. As a result, our management system can be easily integrated with the Server-Load-Blancer from other vendors by extending the control interface to support the proprietary configuration functions of each vendor.

We then extended the functions of remote console for the administrator to configure the system. With the remote console applet, adding or removing a node is an easy job and does not require the extensive reconfiguration of all other nodes. The GUI of remote console applet supports tracking and visualization of the system's configuration and state. As a result, although the system configuration may change dynamically and availability of nodes is also subject to change, the administrator still can easily know which node is operating as a part of the system.

We also provide the monitor mechanism for the administrator to monitor the status (e.g. resource utilization) of each node, ensure the resources provided by the web site are operational, and verify the web content can be delivered normally. Otherwise, some problems that are trivially found in a single system may be hidden for a long time and difficult to be detected. The system will dispatch the managelets to execute on the managed node, collect a variety of statuses, and then report the summary to the remote console after consolidating the raw data. Reliable operational status

of providing hosting service on the distributed server, unless made in such an automated way, requires significant human effort to achieve in such a complex system. In addition, the remote console also provides log and alarm functions to enable the administrator to identify security problems or other situations.

3.4 Content Management

We implemented several new management functions and system calls for the administrator to configure the URL table. The URL table is initialized by a program that parses the content tree of the Home server. We extended the remote console to visualize the URL table, which produces a single, coherent view of the Web document tree that is actually partitioned on different nodes. The remote console provides a file manager interface containing methods for inserting, deleting, and renaming files or directories. With the GUI, the administrator can easily assign different content to different servers for meeting the performance requirements of different customers. The administrator also can assign some specific content to multiple server nodes for fault tolerance or high availability. Whenever the administrator changes the document deployment, the remote console will inform the controller of these changes. The controller will change the URL table to adapt these changes, and send the managelets that perform the content management functions to propagate these changes to the whole system. As a result, although the content deployment may change dynamically, the administrator still can easily know the system's status.

We implemented several managelets to perform the content management functions. For example, one managelet is responsible to add a file into the local file system of the node that it executes. If one spare node is recruited into the server system, the managelet is sent to this node to automatically replicate some content to this node. Similarly, if the node is excluded from the system, one specific managelet is sent to offload web pages from this node. Another example is that we implement a managelet to roam in the system for checking content consistency.

4. High Reliability

In this section, we will describe how a higher reliability could be achieved by our system. Our system guarantees service reliability at tree levels. First, a status detection mechanism can detect and mask the server failures. Second, a request-failover mechanism enables an ongoing Web request to be smoothly migrated and recovered on another server node in the presence of server failure. Third, we implemented a mechanism to prevent the single-point-of-failure problem.

4.1 Failure Detection

The broker, running on each back-end server node, can also be used to provide a failure detection service for the entire system. Periodically (the interval is adaptable), the monitor thread of the broker will wake up and initiate a request to the web server running on the managed node. For minimizing the additional workload added on the managed node, such a request is designed for retrieving a small file. If the server responds normally, the broker sends an "I-am-alive" message (i.e., heartbeat message) to the dispatcher. In addition, the monitor thread also measures the response time of the issued request over each interval. If the new measured value is larger/smaller than the previous measurement, the next interval time will be increased/decreased. The increased/decreased value is proportional to the difference of the two measured values.

There are two main purposes in such a design. First, it will prevent the monitor thread from burdening the load of the web server. If the server is overloaded, the monitor thread will decrease the frequency of the probe by discovering the longer response time. Second, fine-grained load balancing can be achieved by such a design. That is, if the managed server node is overloaded, the time of interval will be lengthened. This means that the broker will increase the time of interval between this heartbeat and the next. The dispatcher keeps a counter for each server node, and such a counter will be incremented periodically. When the dispatcher receives a heartbeat from one server node, it resets the counter associated with this server node. On selecting a server for a new arriving request, the dispatcher will check the counter associated with the candidate server. If the counter exceeds a "warning value", the server node may be either overburdened or unreachable. Such a node will be skipped, and the request will automatically be allocated to the next most available server. As a result, the dispatcher could detect the overloaded node and stop dispatching new requests to it. If the counter exceeds a "dead value" (which is a higher threshold than warning value), the node will be declared dead and be temporarily removed from the server cluster, and an alarm message will be sent to the administrator. As a result, the failed server node can be detected and masked by the whole system.

4.2 Failover

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 failure. We divide web requests into three types: requests for

static content, requests for dynamic content, and requests for session-based services. We devised corresponding solution for each category. We implemented a program in the management system to parse the content tree and specify the type of each Web object in the URL table. For example, files ending with '.jpg' or 'html' are classified into the type of static content, and any URL that include 'cgi-bin' or file ending with '.php' are considered to be a dynamic content. As a result, the dispatcher can identify the type of each request via consulting the URL table, and then failover the requests of each category with the corresponding approach in case of server failure.

4.2.1 Requests for Static Content

To requests for static content, we use the following mechanism to failover a request on another node. First, the dispatcher will select a new server, 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. 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 can 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 reusing pre-forked connection and seamlessly relaying packet between two TCP connections, we can smoothly recover a request on another node.

4.2.2 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. We cannot use the above approach to recover a dynamic request because the result of two successive requests with the same arguments may be 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.

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. To relieve 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.

4.2.3 Requests for Session-based Services

A so-called session consists of a number of user interactions, i.e., 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. To recover a session is important because it is widely used in the E-commerce services.

However, recovering a session on another node is a more challenging 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. It also needs a mechanism to replicate the intermediate processing-state in order to ensure the fault-tolerance of the session itself. We tackle these problems by the following mechanisms.

First of all, we model the session-based services by a state machine that consists of the following states: Start, Browse, Search, update the state, Pay, and End. The web site manager should define a session for which fault resilience or higher performance is required, by specifying some important Web pages a corresponding state. 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 Start of a session; and define the action, "when user clicks the *check-out* button on a specific page", as the End of this session. The administrator could easily make such configurations via the GUI of our management system.

Such configuration information will be stored in the 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 provide the fault tolerance support for all consequent requests, until it finds a request conveying the "end" action. We design a primary-backup protocol [21] to replicate the intermediate state of a session on a backup node. When the primary server that is responsible for processing the session fails, the backup server can take over its job with the replicated state. The readers are referred to [21] for further details.

4.3 System Robustness

We noticed that the dispatcher represents a single-point-of-failure in our system, i.e., failure of the dispatcher would bring down the entire Web server. To improve the robustness of our system further, we can use multiple dispatchers to cooperate for distributing requests. In this configuration, the DNS approach [22] 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. 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.

5. System Evaluation

This section presents the results of a performance study on the prototype system. Due to space limitation, we focus the discussion on the performance experiments that evaluate the benefit of our URL formalization mechanism and URL table. We will report the performance data and a detailed analysis of the fault tolerance mechanism in [21].

5.1 Measurement Setup

We constructed the following server cluster in 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 128MB) machines running Linux with Apache (version 1.2.4), and six Pentium-2 (300 MHz CPU with 128 MB memory) running Windows NT with IIS 4.0. The reason for such a software configuration is that we want to show that our mechanism can operate with any kind of operating system and server software. The Apache servers are responsible for providing static content and session-based service, and the IIS servers are responsible for providing dynamic content. 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 [25] 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. We also inserted the session model into the workload so that session-based service could be investigated. We implemented [21] a session generator in the WebStone client to issue session-based requests. The content hosted in the cluster system consisted of 97 Web sites (with approximately 72000 unique files of which the total size is about 1312MB).

To quantify the benefit of the new content-aware routing mechanism, we measured and compared the response time and throughput in the prototype system equipped with the new mechanism, with those in a baseline system without the proposed mechanism. The baseline system is a server cluster (with the same configuration described above) front-ended by the dispatcher implementated in our previous work [11]. The dispatcher in the baseline system needs to parse the entire HTTP header to find out the host field for making routing decision.

5.2 Results

The peak throughput of the proposed system is 3278 requests/sec. In contrast, the peak throughput of our previous system is 2365 requests/sec. At the period of peak throughput, the CPU utilization of the dispatcher

equipped with the new mechanism is 52% (and the previous is 78%). The results show a significant performance improvement.

The reason for this higher performance is because of the clever design of URL formalization and its associated data structure. The dispatcher can quickly identify that the incoming request is for which Web site, rather than parse the entire HTTP header to find out the host field. Combined with the well-designed URL table, the dispatcher can quickly retrieve related information to make routing decision. To quantify the benefits of such a design, we instrumented the Linux kernel of the dispatcher to measure the latency of URL parsing and searching. We generated a heavy load (128 clients) to push the server into a prolonged overload state, and then we measured the processing time (i.e., the time of parsing a HTTP header and the time of searching the URL table to retrieve the routing information).

As we described above, the content hosted on the test system contained about 72000 Web objects. In such scale, the memory consumed by the URL table is about 1.8 Mbytes. During the peak load, the average processing time on a HTTP request is about 2.14 μsec. In contrast, the average processing time in the baseline system is about 248 μsec.

The major concern of a system equipped with some fault-tolerance mechanisms might be how much overhead is associated with these mechanisms, and if the system's performance will suffer from the additional overhead. The higher throughput of the proposed system can demonstrate that the performance concern does not exist in our system. To precisely quantify the additional overhead, we analyzed and compared the response time of a Web request in our system, with those in a baseline system without fault-resilience support. Compared with the average response time of the requests in the two configurations, we could quantify how much additional overhead will be introduced by the fault resilience mechanism.

The results of requests for static content are given in Table 1. We do not see any performance degradation introduced by the fault resilience mechanism. We also found a similar result in the performance data regarding the dynamic content. The reason of the low overhead is that we do not need to keep any additional state for recovering a request for static and dynamic content.

Request size (Kb)	4K	8K	32K
Our system (ms)	24.89	34.53	171.93
Baseline (ms)	23.58	32.25	170.24
Request size (Kb)	**64K**	**256K**	**1024K**
Our system (ms)	305.29	1147.42	4809.14
Baseline (ms)	308.39	1145.62	4815.17

Table 1 Overhead (Static content)

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. 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 comparing with the typical latency over the wide-area networks [26,27].

6. Discussion

In this section, we discuss the advantages of our system and possible arguments against it. We also describe the related work.

6.1 Advantages

The first advantage of our system is that we provide an innovative management system to address the management problem of deploying web hosting services on a server cluster. With the management system, hosting service providers can easily manage and maintain content on the distributed server as a single large system. The management system provides the administrator with a single system image on system management. The system can also automate many management operations that could ease the burden of system management. In addition, our management system offers a natural concurrent-problem-solving paradigm, which provides a scalable solution to some tedious management tasks in a large distributed server. For example, we implemented a managelet to retrieve some specific URL to determine the availability of the content. The administrator can configure and dispatch the managelet to several nodes to perform content-level health checks concurrently.

The second advantage of our system is that we provide a higher level of service-reliability support than other server cluster systems. While the other systems can only provide high availability by masking the server failures, our system can provide high reliability by enabling the requests on the failed node to be recovered on another working node. Such a capability is important and essential for the E-commerce providers. They are willing to pay higher fee to get the guarantee of high service reliability, since they know that service outage in today's highly competitive marketplace can mean lost revenue and lost credibility. Our content-aware routing mechanism can enable the Hosting service providers to charge for differentiated services. The dispatcher can identifies the type and importance of each request, and then provides corresponding level of reliability support.

HAWA [3] 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. [28] and Chawathe et al. [29] proposed a framework for building a highly available Internet service on the cluster system. These systems indeed could provide higher availability, however, they are not fault-tolerant or 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. Ingham et al. [30] 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.

The idea of content-aware routing is not new. In addition to us, a number of research projects [9,10,31] and commercial products [e.g., 32-40] also use a similar idea. However, we make the following unique contributions. First, we augment the content-aware routing with the fault-resilience capability. Second, we devised the new idea of URL formalization and a corresponding data structure. Comprehensive content-specific knowledge can be stored in the table, from self-learning by the management system or administrator's inference. Without the support of the content management system, configuring and managing the URL table will be a difficult problem. In contrast, the data structures and what kind of content-aware intelligence used in these commercial products are unclear, at least, have never been described clearly in the literatures.

Finally, the request migration technique not only can be used to provide fault resilience; it is also beneficial to system management. If we need to perform several management actions on one server node, and that might overburden this node, the system can automatically "migrate" the ongoing requests on this node to other nodes. Otherwise, if we want to temporary remove a node for maintenance, the request migration mechanism can be invoked to transfer the ongoing requests to other nodes. The same technique can also be used to solve the "flash crowd" problem. It is well known that certain event on particular web site could trigger a significant load burst that persists for hours, or even days. Examples include announcement of a new version of popular software or product, a site mentioned as the "best-site-of-the-week" on the news, or political sites during a campaign. Although the clustered server can provide compelling performance and accommodate the growth of web traffic, it could suddenly become swamped due to receiving far more requests on one node than it was originally configured to handle. The request migration mechanism can be invoked in response to server overloaded.

6.2 Possible Arguments Against our System

Someone might argue that the content management problem could be solved by a shared file system. Through the communication network, server nodes at different locations can access the content from the shared file system to serve user requests. The advantage of this approach is that we can easily manage and maintain the content with a centralized policy. However, such a design will suffer from the following problems. First, accessing data over the network file system will increase user perceived latency due to the overhead of remote-file-I/O and LAN congestion. Some specific distributed file systems (e.g., AFS [41]) that provide cache mechanism might alleviate this problem. However, it is well-known that the access patterns of WWW exhibit high skew with regard to the access frequencies of content, thus the popular files tend to occupy RAM space in all the nodes. The redundant replication of "hot" content through the RAM of all the nodes leaves much less of available RAM space for the rest of the content, leading to a worse overall system performance. Second, the shared file system approach does not take the variety of content into consideration.

6.3 Work in Progress

We are pursuing on extending and completing the functionality of the system in the following aspects. First, we are designing a load balancing mechanism to ensure an even load distribution in the system. Since the access patterns of WWW exhibit high skew, the static content-placement may lead to load imbalance. In other words, the servers that store the popular documents will get overloaded, resulting in hot spots and bad system performance. As a result, we are implementing an auto-replication facility to dynamically adjust content placement for ensuring an even load distribution. Cherkasova et al. [42,43], Narendran et al. [44], Rabinovich et al. [45,46] have done great works in a similar problem.

Second, we are investigating the issue of service Level agreement (SLA), which will enable the content owners to specify their specific requirements such as bandwidth usage, number or placement of content replicas, or required degrees of service reliability. With the proprietary request-failover mechanism, our system can offer a strong SLA (service-level agreement) on service reliability: the important customers can be promised that no user requests will be lost due to server overload or failure. We are implementing the related mechanisms to configure the management policy to meet the complex requirements of different customers.

Third, we are investigating how to support service differentiation and QoS guarantee to satisfy the customer's requirement.

7. Conclusion

Business demand is fueling the market for companies that specialize in hosting and managing other companies' Web sites. However, there are lots of actions and functions in most hosting service providers at the low end today. They must gradually move to more sophisticated services as the content of a Web site and e-business operations become more complex. In this paper, we have described the research work we are pursuing in this direction. We provide an integrated framework to construct a reliable and highly manageable Web hosting service on a scalable server cluster. We have demonstrated that the URL Formalization mechanism and a corresponding data structure can provide a scalable solution to the request distribution in the system. The management system can mask the complexity of such a distributed environment, providing a highly manageable system. A higher level of services reliability can be achieved by the proposed system. We believe that we have taken an important step toward providing a successful hosting service.

Availability

You can get more information about this system and see a demo demonstration of the remote console on our Web site: http://pds.cse.nsysu.edu.tw.

Acknowledgement

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

References

[1] A. Fox, S. Gribble, Y. Chawathe and E. A. Brewer, "Cluster-based scalable network services," In Proceedings of SOSP '97, October 1997.

[2] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler, "Using smart clients to build scalable services," In Proceedings of the 1997 USENIX Annual Technical Conference, January 6-10, 1997.

[3] Y. M. Wang, P. Y. Chung, C. M. Lin, and Y. Huang, "HAWA: a client-side approach to high-availability web access," In Proceedings of the Sixth International World Wide Web Conference, April 1997.

[4] R. McGrath T. Kwan and D. Reed, "NCSA's World Wide Web server: design and performance," IEEE Computer, November 1995.

[5] V. Cardellini, M. Colajanni, P.S. Yu, "DNS dispatching algorithms with state estimators for scalable Web-server clusters", World Wide Web Journal, Baltzer Science, Vol. 2, No. 3, pp. 101-113, Aug. 1999.

[6] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A scalable and highly available web server," In Proceedings of the COMPCON'96, February 1996.

[7] G. D. H. Hunt, G. S. Goldszmidt, R. P. King and R. Mukherjee. "Network dispatcher: a connection router for scalable Internet services," In the Proceedings of the 7th International World Wide Web Conference, April 1998.

[8] D. Andresen, T. Yang, O. Ibarra, "Towards a scalable distributed WWW server on networked workstations," Journal of Parallel and Distributed Computing, Vol 42, pp. 91-100, 1997.

[9] V. Pai, M. Aron, M. Svendsen, G. Banga, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[10] A. Cohen, S. Rangarajan, and H. Slye, "On the performance of TCP splicing for URL-aware redirection," In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October 11-14, 1999.

[11] C. S. Yang and M. Y. Luo, "Efficient support for content-based routing in Web server clusters," In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October 11-14, 1999.

[12] J. C. Mogul, "The case for persistent-connection HTTP," In Proceedings of the SIGCOMM'95 August 1995.

[13] T. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. C. Mogul. Hypertext Transfer Protocol–HTTP/1.1--Draft Standard RFC 2616, June 1999.

[14] The Advantages of F5 Layer 7 Management, White paper of F5 Lab**.** Available at http://www.f5.com/solutions/whitepapers/layer7.html

[15] A. Iyengar, E. MacNair and T. Nguyen, "An analysis of Web server performance," In Proceedings of the GLOBECOM '97, November 1997.

[16] K. Arnold and J. Gosling. The Java Programming Language. Addison-Wesley Publishing Company, Reading, MA, 1998.

[17] C. S. Yang and M. Y. Luo, "Design and implementation of an administration system for distributed Web server," In Proceedings of the 12th USENIX Systems Administration Conference, December 1998.

[18] L. Gong. Inside Java 2 Platform Security, Addison Wesley, Reading, MA, June 1999.
[19] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias, “Design and performance of a Web server accelerator,” In Proceedings of the INFOCOM'99, March 1999.
[20] J. Challenger, A. Iyengar, and P. Dantzig, “A scalable system for consistently caching dynamic Web data,” In Proceedings of the INFOCOM'99, March 1999.
[21] M. Y. Luo and C. S. Yang, “Constructing Zero-loss Web Services,” In Proceedings of the INFOCOM 2001, April 22-26, 2001.
[22] M. Colajanni, P.S. Yu, D.M. Dias, “Analysis of task assignment policies in scalable distributed Web-server systems”, IEEE Trans. on Parallel and Distributed Systems, Vol. 9, No. 6, June 1998.
[23] Y. Huang and C. M. R. Kintala, “Software implemented fault tolerance: Technologies and experience,” In Proceedings of 23rd Intl. Symposiumon Fault-Tolerant Computing, pages 2–9, Toulouse, France, June 1993.
[24] Y. Huang et.al., “NT-SwiFT:Software implemented fault tolerance on Windows NT,” In Proceedings of the 2nd USENIX NT Symposium, Seattle, August 1998.
[25] WebStone, http://www.sgi.com/
[26] P. Barford and M. E. Crovella, “Measuring Web performance in the wide area,” Performance Evaluation Review, August 1999.
[27] M. Kalyanakrishnan, R. K. Iyer, and J. U. Patel. “Reliability of Internet hosts: a case study from the end user's perspective,” Computer Networks, 31, pp. 47-57, 1999.
[28] A. Singhai, S.-B. Lim, and S. R. Radia, “The SunSCALR framework for Internet servers,” In Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, June 1998.
[29] Y. Chawathe and E. A. Brewer, “System support for scalable and fault-tolerant Internet services,” In Proceedings of Middleware '98, September 1998.
[30] D. Ingham, F. Panzieri, S.K. Shrivastava, “Constructing dependable Web services,’ IEEE Internet Computing, Vol.4, N. 1, January/February 2000.
[31] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, D. Saha, “Design, implementation and performance of a content-based switch,” In Proceedings of the Infocom 2000, March 2000.
[32] Alteon. Alteon 180 Web Switch. http://www.alteonwebsystems.com
[33] ArrowPoint. CS-100 Switch. http://www.arrowpoint.com/
[34] Coyote Point. Equalizer. http://www.coyotepoint.com
[35] F5Labs. Big/IP. http://www.f5.com/
[36] Foundry Networks. ServerIron. http://www.foundrynet.com.
[37] Lucent. WebSwitch. http://www1.bell-labs.com/project/webswitch/default.htm
[38] HydraWeb. Hydra 5000. http://www.hydraweb.com
[39] IPivot. http://www.intel.com/network/ipivot/index.htm
[40] Resonate, http://www.resonate.com.
[41] M. Satyanarayanan. “Scalable, secure, and highly available distributed file access,” IEEE Computer 23, 5 May 1990.
[42] L.Cherkasova, “FLEX: load balancing and management strategy for scalable Web hosting service,” In Proceedings of the Fifth International Symposium on Computers and Communications, July 3-7, 2000.
[43] L. Cherkasova and S. Ponnekanti, “Achieving load balancing and efficient memory usage in a Web hosting service cluster,” HP Laboratories Report No. HPL-2000-27, February 2000.
[44] B. Narendran, S. Rangarajan and S. Yajnik, “Data distribution algorithms for load balanced fault tolerant web access”, In Proceedings of the Symposium on Reliable and Distributed Systems, October 1997.
[45] M. Rabinovich and A. Aggarwal, “RaDaR: a scalable architecture for a global Web hosting service,” In Proceedings of the 8th International World Wide Web Conference, May 1999.
[46] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal, “A dynamic object replication and migration protocol for an Internet hosting service,” In Proceedings of the IEEE International Conference on Distributed Computing Systems, May 1999.