ADAPTIVE LOAD BALANCING AND CHANGE VISUALIZATION FOR WEBVIGIL

by

Subramanian Chelladurai Hari Hara

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

DECEMBER 2006

To my Mother, my Father, my Teachers and God

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr.Sharma Chakravarthy for constantly motivating and encouraging me, and also for his invaluable advice during the course of my masters studies. I wish to thank Dr.David Kung and Dr.Mohan Kumar for their interest in my research and for taking time to serve in my defense committee.

I am especially grateful to Dr.Raman Adaikalavan for his continuous support, motivation and encouragement. I am grateful to all the teachers who taught me during the years I spent in school, first in India, then in Kuwait and finally in the Unites States.

This work was supported, in part, by NSF (grants IIS-0123730 (webvigil), IIS-0326505 (PSI), and EIA-0216500 (MRI))

Finally, I would like to express my deep gratitude to my father and mother for their sacrifice, encouragement and patience. I am extremely fortunate to be so blessed. I am extremely thank full to all ITLAB'ians, friends and roommates (Mr.Aravind Venkatachalam, Mr.Aravind Vummidi and Mr.Suresh Kumar) for their moral support.

I am also thank full to Mr.Bito Irie for giving me an opportunity to serve the CSE Department as System Administrator for Linux and Windows systems which supported me during the course of my thesis.

November 27, 2006

ABSTRACT

ADAPTIVE LOAD BALANCING AND CHANGE VISUALIZATION FOR WEBVIGIL

Publication No. _____

Subramanian Chelladurai Hari Hara, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Dr. Sharma Chakravarthy

There is a need for selective monitoring the contents of web pages. Periodical visit for understanding changes to a web page is both inefficient and time consuming. WebVigiL is a system developed for automating the change detection and timely notification of HTML/XML pages based on user specified changes to the contents of interest. User interest, specified as a sentinel/profile, is automatically monitored by the system using a combination of learning-based and event-driven techniques.

The first prototype concentrated on the functionality of the WebVigiL system. This thesis extends the WebVigiL system in a number of ways. The primary focus of this thesis is to improve the performance and scalability aspects of the prototype. A load balancer is proposed based on the analysis and estimation of the load factors imposed by a sentinel on the system. An adaptive load balancer has been designed for WebVigiL that uses various

properties to distribute sentinels among servers using a number of strategies. These strategies have been implemented and experimentally compared with analysis. A web-based `dashboard` has been developed to enable users to manage and visualize changes to their sentinels. As part of the `dashboard`, a `Dual-Frame` presentation for displaying the detected changes is provided both for HTML and XML. Other contributions of the thesis include integration of the above modules into the current system, making the WebVigiL system stable and robust (by fixing various bugs) and testing the system for various test cases.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Evolving Change Detection

In recent times, the synonym to global information space is nothing other than the World Wide Web. It contains text documents, images, multimedia and many other types of information, referred to as resources, presented together as a web page, identified by A unique global identifier called Uniform Resource Identifier (URI). In addition to growth in the number of web pages, the content of the pages on the web is also changing continuously. Users may be searching for specific information or interested in changes to specific web pages. For example, researchers might want to know which conference has extended the deadline for paper submission and the date of the new deadline. They might also want to track if any new papers have been added to A faculty'S web site. As another example, students might want to track their course web site for updates on home work, projects, or assignments. Periodical retrieval of pages for understanding changes is both inefficient and time consuming. In general, the ability to specify and monitor for changes to arbitrary web pages and get notified in a timely manner is necessary and is definitely a more effective and efficient alternative to users' visiting the web page periodically for monitoring the changes.

1.2 Existing Paradigms

Change detection for web content can be done in one of two ways: The pull paradigm [1] and the push paradigm [1] approach. The first approach entails posing appropriate query to the server (where the page of interest resides) for selective changes.

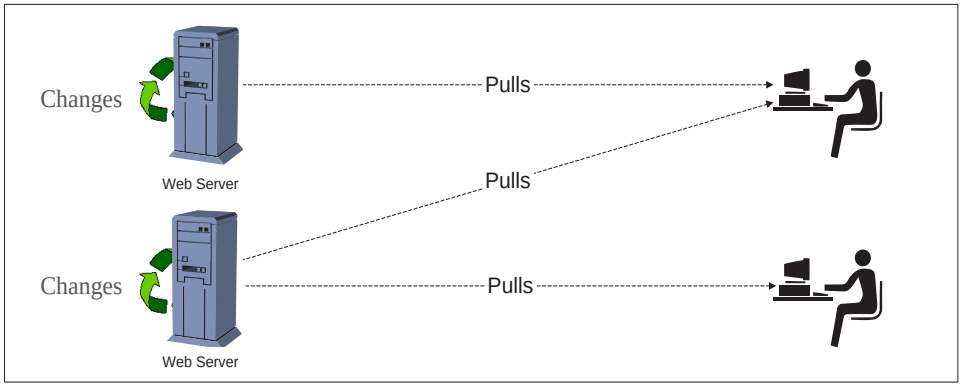

Figure 1.1 Pull Paradigm

In the second approach the server pushes the information of interest to the subscribed users.

1.2.1 Pull Paradigm

The user is responsible for constantly monitoring the web page of his/her interest for changes. In other words, the information of interest is pulled from its source by the user. Figure 1.1 clearly indicates the pull paradigm. The frequency with which the user can pull the pages to see a change is also not fixed. It is a labor intensive approach and requires human in the loop. This approach cannot be used if the page changes frequently as there is a significant overhead involved in constantly retrieving the page for identifying specific changes.

1.2.2 Push Paradigm

The users are notified of the changes once they subscribe to the website of their interest for alerts. The alerts are sent to the users using mailing lists. Figure 1.2 indicates the idea of sending out notifications to clients in a timely manner. This approach reduces the burden on users. But the user has to be satisfied with whatever information the server

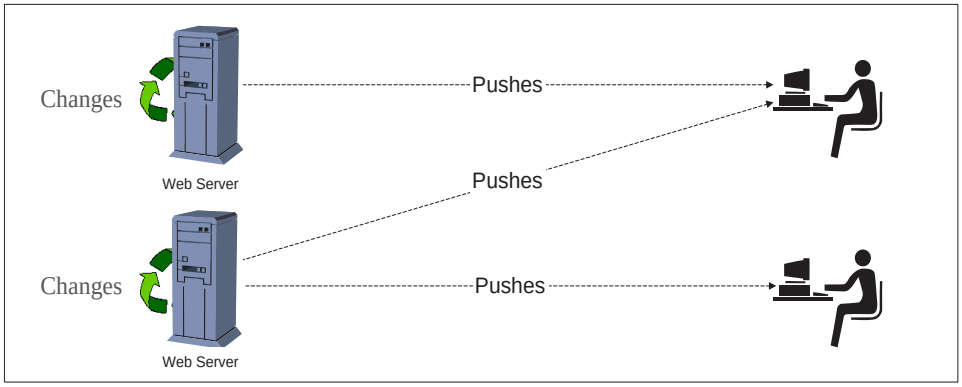

Figure 1.2 Push Paradigm

sends them. This approach can be used if there are frequent changes, but there is also a overhead involved in sending out change notifications to clients.

1.3 Motivation

In order to get the benefits of both the approaches, we can use a third approach where a middle agent pulls information intelligently and transparently from the server and pushes the information that is relevant to the user [1]. This approach is used by WebVigiL [2], the users can subscribe for a change of interest on a page, for which they receive customized notification. Figure 1.3 shows pictorial representation of WebVigiL's approach.

WebVigiL is a general-purpose, web page monitoring and notification system which monitors HTML/XML pages according to user-defined profile (termed sentinel). Users' place a request by giving a set of specifications, such as the page to be monitored, the type of change to be checked for, presentation of changes, and how to notify the changes. The system monitors the specified pages and notifies the users of the changes consistent with

[1] since current web servers do not push information and are not open source, This is a realistic alternative. Potions of the logic of the middle-ware can certainly be pushed into web servers, if allowed.

their specification. WebVigiL uses active capability to provide timely responses. Event-Condition-Action (or ECA) rules are used to provide active capability to the system. In ECA rules, on the occurrence of an event, condition associated with that event is evaluated and associated actions are performed if the condition evaluates to true. In WebVigil, primitive events and composite events are mapped to corresponding change in A page such as change to images, links, keywords, Phrases etc. ECA rules are also used for creation of a sentinel, monitoring the requested page, notifying the user of the change and deactivation of a sentinel.

Since WebVigiL is a web-based system that can be used by anyone from anywhere (just like GOOGLE), there can be situations where it has to accommodate a large number of sentinels. In order to provide service to all the clients in A timely manner, it is necessary to keep WebVigiL running by reducing the amount of load experienced by the system when more sentinels comes in. In order to handle such situations, this thesis concentrates on finding those load parameters that affect the performance of the system, estimate these load parameters using the sentinel description and distribute the load among several servers in a server farm using various strategies. The web-based user interface of WebVigiL allows the clients to set sentinels, enable/disable them, and view the detected changes in a more visually pleasing and easy to understand manner. The interface also allows the users to track their sentinels and view when a change was detected and what the changes were. This thesis improves upon the presentation of detected changes. The dual-frame visualization of changes in HTML and XML That was proposed earlier has been extended and implemented as part of this thesis. The final part of the thesis addresses enhancements to the HTML parser to handle dynamic pages, new type of tags, and content presentation technique. The new parser IS able to handle pages which were causing the old parser to run out of memory or truncate the content before parsing, thus improving accuracy of change detection.

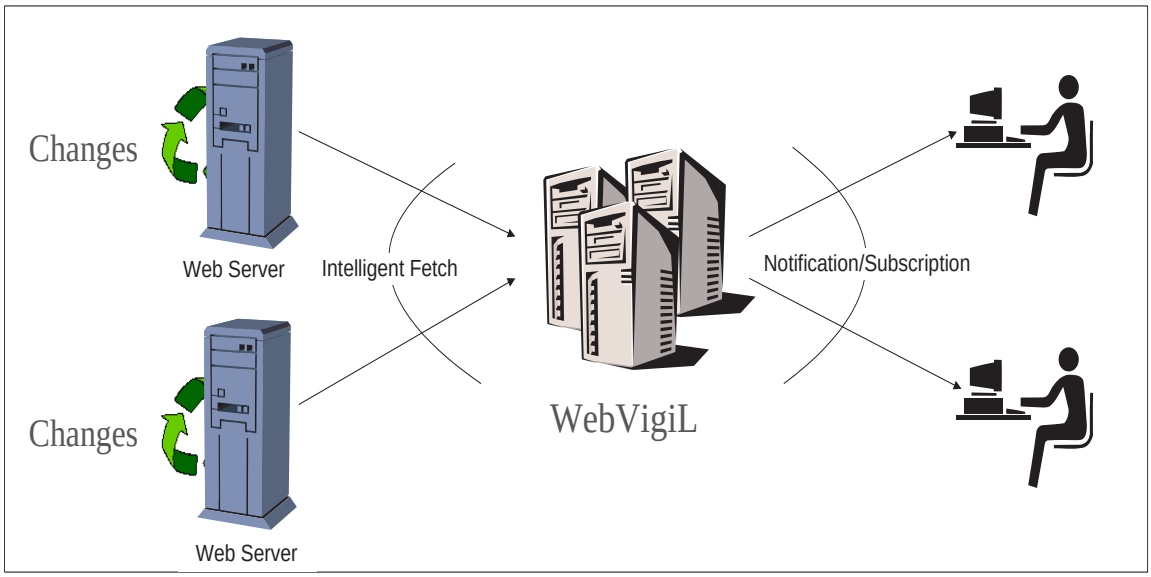

Figure 1.3 Intelligent Push/Pull in WebVigiL

1.4 Contributions

The desired functionality of WebVigiL was achieved using the modules developed in previous theses [3, 4, 5, 6, 7, 8]. Since there was a need to enhance the performance of the system by scaling the system, it required analyzing the present system to identify the areas where performance needed a boost. Further the performance parameters were taken into consideration to develop a distributed version of the system. Thus the objective of scalable WebVigiL system was achieved. Highlights of contributions that forms the focus of this thesis are discussed below.

The first contribution of the thesis is The development of formulas to determine the load imparted by the sentinels on the WebVigiL server and distribute the sentinels among the servers in the WebVigiL server farm. The improved load balancing will allow WebVigiL to be scaled and distributed. The next contribution is the development of a DashBoard, where users will be able to manage and maintain their sentinels in addition to viewing changes. The DashBoard gives the users the ability to enable/disable sentinels

and even reduce the number of notifications they receive. The third contribution is to develop algorithms for presenting the changes in A color-highlighted dual-frame format. The final contribution is to integrate the system and replace the current HTML parser with a new one to handle dynamic pages, pages with new type of tags, and content presentation.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 describes some of the related work that has been done in the area of monitoring web content and Server Load Balancing. Chapter 3 gives an overview of WebVigiL architecture. Chapter 4 discusses design and development of load balancer which makes WebVigiL a scalable system. Chapter 5 discusses the architecture and implementation details of the load balancer. Chapter 6 discusses the design of DashBoard (Web-Based user interface for WebVigiL), improvements in the presentation algorithm and how accuracy of change detection is improved. Finally Chapter 7 outlines conclusions and future work.

CHAPTER 2

RELATED WORK

In this chapter, some of the related work in the field of change detection over content present on the Web are discussed. Also, the earlier works on the scope of Server Load Balancing are discussed, which are the main contributions of this thesis.

2.1 Change detection systems

2.1.1 WebCQ

WebCQ [9] is a prototype system for large-scale web information monitoring. It uses the concept of continuous queries to perform change detection. WebCQ is designed to detect and notify interesting changes to users with personalized customization. The current WebCQ system supports change detection over various primitive change types such as links, images, keywords, lists, tables but there is no provision to give composite change requests. Also, the current WebCQ system does not support the monitoring of multiple web pages with a single request. User's cannot combine multiple URLs with different change types. The frequency with which the web page should be fetched for performing change detection is given by the user. There is no provision to request the system to tune the fetch frequency to the actual change frequency of the web page. There is no feature to give dependent requests, wherein a request can be given based on a previously given request. WebVigiL provides the functionality to monitor multiple web pages with a single request. It also provides the feature to monitor a web page and all the web pages linked from it. WebVigiL also has a learning-based fetch mechanism to fetch web pages based on the history of observed changes. One advantage of the system

is that it supports the monitoring of lists and tables which WebVigiL does not support. Also, WebCQ does not address Load Balancing issues. It does not provide the equivalent of Dual-Frame presentation as is done in WebVigiL

2.1.2 Commercial change detection systems

- **Watch That Page:** Watch That Page [10] can monitor an unlimited number of pages, which can be grouped into folders and monitored on a daily or weekly basis. There is a keyword matching option that filters the changes that are relevant to the users. Channels enable users to divide the pages into groups based on importance or content type. Each channel can have different properties: some can have keyword matching and daily reports while others can be checked less frequently and report all changes. Email alerts can include the text that has changed on users pages or just list the URLs of pages that have changed.
- **Wisdomchange:** Wisdomchange [11] is a free service that checks for changes to the text of a web page. Changes in HTML tags or images are ignored. In addition to the standard email alerts users can also have notifications sent to their mobile phone or pager. There is no limit to the number of bookmarks that users can set up.
- **Change Detection:** changedetection [12] can monitor unlimited number of pages but there is no option on the site that allows users to view all of their set pages. This means that user can easily lose track of what they are monitoring. There are no options for changing the frequency of monitoring - it seems to be done on a daily basis - or for selecting text or keywords. The email merely notifies user that the monitored page has changed. The notification includes a link to the changed page and an option to "un-subscribe".

- **ChangeDetect:** Changedetect [13] allows users to monitor a maximum of 5 pages a week and marks web page text for user with color-coded highlights of what has changed. Users can also receive change notifications via their email, pager, ICQ or text messaging. The subscription service allows users to monitor more pages and password protected pages.
- **Track Engine:** Trackengine [14] allows setting up an easy to remember name or description and the monitoring frequency (daily, 2 days, 3 days or weekly) for each page. The advanced options enable users to select the color for highlighting the changes on the monitored page, and to track changes to hyper links, images, numbers and dates as well as the text. Users can request alerts for changes that include specified words or phrases and tell it to ignore certain changes. Users can even monitor pages that are password protected by supplying Track Engine with user name and password.
- **Copernic Tracker:** Copernic [15] once installed on users PC, enables users to track any number of pages on external sites and intra nets. Users can track changed words, new links or images. There is a useful advanced query form for tracking specific words within pages, Boolean and other search operators (AND, OR, NEAR). Copies of page revisions are stored locally so that users can compare changes that occurred in the past and add their own notes for tracked pages and each of their revisions. There is an option for importing favorites from Internet Explorer so that users don't have to key in lists of sites, but no facility for directly importing Mozilla/Netscape bookmarks or Opera Hot lists. There are four pre-set tracking schedules: Multiple Times per Day, On a Daily Basis, On a Weekly Basis and On a Monthly Basis. Alerts can be a tray icon, desktop alert or notification message, SMS notification, email report with the tracked page contents and changes highlighted.

- **Website Watcher:** Aignes [16] monitors an unlimited number of pages and users can choose to ignore HTML tags, images/banners, numbers and dates. User enters user names and passwords for password protected pages that they wish to monitor. Pages can be checked once a day, once a week or on a specified day/days of the week. This is the only service that enables users to monitor entire sites without having to specify each page individually. There is an option that allows user to specify the checking frequency during a day either in hours or minutes.

2.2 Server Load Balancing

If there is only one server responding to all the incoming requests from users, the capacity of the server may not be able to handle high volumes of incoming traffic. The existing requests will be processed slowly as some of the users will have to wait until the server is free to process their requests. The increase in traffic and connections to server can lead to a point where upgrading the server hardware will no longer be cost effective. In order to achieve server scalability, more servers need to be added to distribute the load among the group of servers, which is also known as a server cluster. The load distribution among these servers is known as load balancing [17, 18]. Load balancing applies to all types of servers (application server and database server).

2.2.1 Types of load balancing

There are several algorithms used to distribute load among the available servers.

- **Random Allocation:** In a random allocation, the user requests are assigned to any server picked randomly among the group of servers. In such a case, one of the servers may be assigned many more requests to process, while the other servers are sitting idle. However, on average, each server gets its share of the load due to the random selection. The main advantage of this approach is that it is simple to

implement and the disadvantage being that it can lead to overloading of one server while under-utilization of others. This allocation also assumes that any server can service any request.

- **Round Robin Allocation:** In a round-robin algorithm, load balancer assigns the requests to a list of servers on a rotating basis. The first request is allocated to a server picked randomly from the group. For the subsequent requests, the load balancer follows the circular order to redirect the request. Once a server is assigned a request, the server is moved to the end of the list. Round-Robin allows for a fair allocation of requests assuming that each request adds the same load to the server. This approach is better than random allocation because the requests are equally divided among the available servers in an orderly fashion, but is not enough for load balancing based on processing overhead required and if the server specifications are not identical to each other in the server group.
- **Weighted Round Robin Allocation:** Weighted Round Robin is an improved version of the round-robin that eliminates the deficiencies of the plain round robin algorithm. In case of a weighted round-robin, a weight is given to each server in the group to determine the load it can take. In such cases, the load balancer will assign proportionately more requests to the powerful server compared to the weaker one, based on the weight ratio. This algorithm takes care of the capacity of the servers in the group, but it does not consider the advanced load balancing requirements such as processing times for each individual request.

2.2.2 Methods of load balancing

There are various ways in which load balancing can be achieved. The deciding factors for choosing one over the other depends on the requirement, available features,

complexity of implementation, and cost. For example, using a hardware load balancing equipment is very costly compared to the software version.

- **Round Robin DNS Load Balancing:** The built-in round robin feature of BIND of a DNS server can be used to load balance multiple web servers. It is one of the early adopted load balancing techniques to cycle through the IP addresses corresponding to a group of servers in a cluster. This method is Very simple, inexpensive and easy to implement, but the DNS server does not have any knowledge of the server availability and will continue to point to an unavailable server. It can only differentiate by IP address and not by server port. The IP address can also be cached by other name servers and requests may not be sent to the load balancing DNS server.
- **Hardware Load Balancing:** Hardware load balancers can route TCP/IP packets to various servers in a cluster. These types of load balancers are often found to provide a robust topology with high availability, but has a much higher cost. Hardware load balancing uses circuit level network gateway to route traffic so it is faster but is not affordable for small scale applications because of its higher costs compared to software versions.
- **Software Load Balancing:** Most commonly used load balancers are software based, and often comes as an integrated component of expensive web server and application server software packages. Software load balancers are cheaper than hardware load balancers: more configurable based on requirements, can incorporate intelligent routing based on multiple input parameters. Sometimes these load balancers have to be run on dedicated hardware to isolate the load balancer.

2.2.3 Commercial server load balancers

- **Cisco LocalDirector:** Cisco System's Local Director [19] is a high-availability, Internet scalability solution that intelligently load balances TCP/IP traffic across multiple servers. Servers can be automatically and transparently placed in or out of service, and LocalDirector itself is equipped with a hot standby fail over mechanism, eliminating all points of failure for the server farm. LocalDirector is a high performance Internet appliance with proven performance in the highest traffic Internet sites
- **F5's BIG-IP:** F5's BIG-IP [20] provides high-availability load balancing, fast and extremely intelligent layer 7 switching, granular interactive control, DoS protection, resource pooling and delivers the fastest traffic management solution to secure, deliver and optimize application performance.

CHAPTER 3

WEBVIGIL ARCHITECTURE

WebVigiL is a profile based change detection and notification system that monitors changes occurring to HTML/XML pages. The purpose of the system is to facilitate selective change detection over the different components of documents (links, images, keywords in HTML documents and keywords and phrases in XML documents). WebVigiL uses various modules to achieve desired functionality. The modules in WebVigiL are mapped to the following functionality of specifying, managing and propagating user requests to monitor web pages at different levels of granularity [21]. The rest of the chapter briefly describes various modules of WebVigiL [2]. Figure 3.1 summarizes the high level WebVigiL Architecture.

3.1 User Specification Module

Monitoring requests are placed in the form of sentinels or profiles. A web-based user interface allows users to submit their sentinels to the system. A sentinel consists of relevant questions which the user answers to specify his/her requirements to a very high level of granularity. A simple sentinel comprises of the following questions:

- Which page needs to be monitored (URL).
- What type of content change needs to be monitored.
- With what frequency the targeted page changes (either user-specified or system-determined)
- When to Start and When to End the monitoring request.
- Where should the notification be sent.

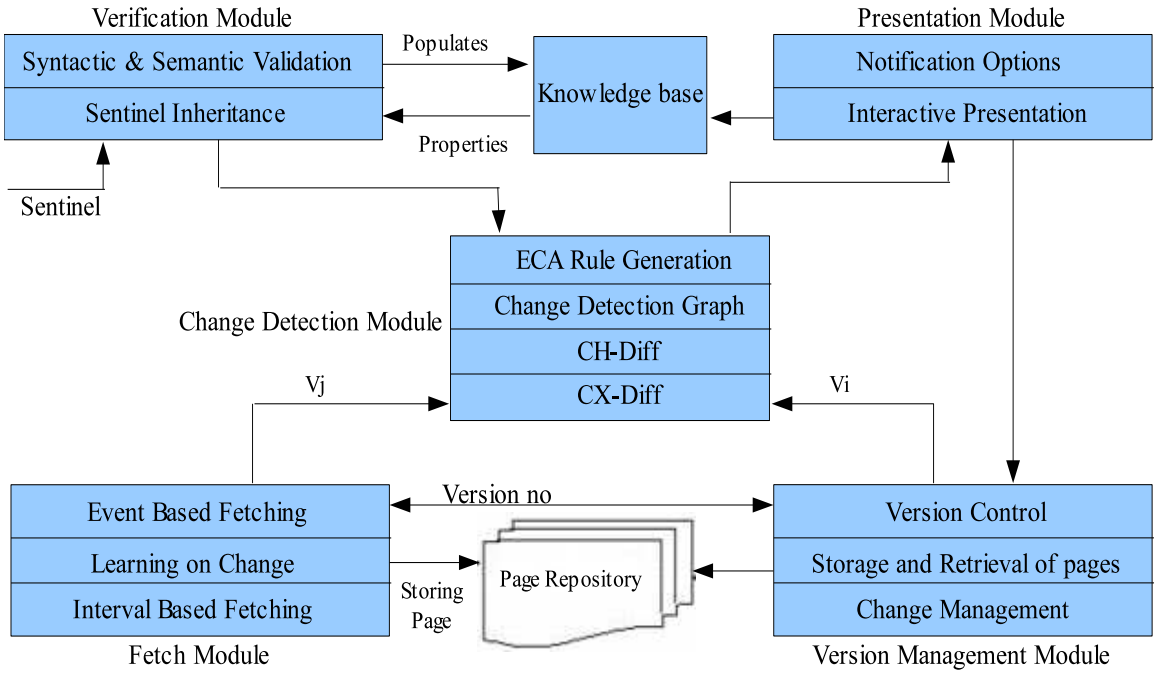

Figure 3.1 WebVigiL Architecture

WebVigiL also supports advanced features for specifying a sentinel, which is internally supported by an expressive specification language [5, 22] with well-defined semantics. The features supported by specification language are summarized below. WebVigiL supports monitoring a page at different levels of granularity such as changes to images, links, specified keywords or phrases. The users can even combine the above mentioned change types by using the logical operators 'OR', 'AND' and 'NOT' (Example: links AND images). The users can set sentinels based on previously placed sentinels which can be used to track correlated changes. In this case, unless the users explicitly specify the properties, the new sentinel inherits all the properties of the previous sentinel. The users can specify the fetch frequency with which the requested page should be fetched and checked for changes. The fetch frequency can either be user-defined (e.g., every 2 days) or system-defined where the system tries to tune the fetch frequency to the actual change frequency using a learning algorithm based on the statistical characteristics of

the history of observed change intervals. The media for notifications can be specified as email or DashBoard. The frequency of notification can be given as user-specified or system-defined frequency. The options for the relative versions of a page to be compared are: moving, every, and pairwise. A sentinel with option moving(n) compares the fetched version of a page with the last n^{th} version in a sliding window. In every(n) every n versions of a page are compared and in pairwise, every version is compared with the previous version. For example consider the scenario: Harry wants to monitor changes to *links and images* of the page *"http://www.rediff.com"* and wants to be notified *daily* by *e-mail* starting from *November 1,2006 to November 29, 2006.* The sentinel specified for the above scenario is as follows:

Create Sentinel s1 Using http://www.rediff.com

Monitor all links AND all images

Fetch every 1 Hour

From 11/01/06 To 11/29/06

Notify By email harry@itlab.com Every 1 day

Compare pairwise

3.2 Validation Module

The validation module is responsible for checking if the input conforms to the syntax and semantics of the specification language. At first the input is validated on the client side (Web Interface) using Java Script. After this, some of the validations are completed on the server side. This certainly reduces the load of communication with the server for all the validations. Only those validations that depend on the information stored on the server are done on the server side. For example the start and end times of inherited sentinels are checked. If the start of a sentinel s1 was specified as the end of another sentinel s2, and at the time of specification if s2 had already expired, the user is

notified of the incompatibility. The sentinel name is also checked for duplication for the same user so that every sentinel installed by the same user should have a unique name.

3.3 KnowledgeBase Module

Validated sentinels details are stored in a repository known as KnowledgeBase. The KnowledgeBase is maintained as a set of relational tables in a database (Oracle). As there are several modules that require the information of a sentinel at run time, it is essential for the meta data of the sentinel to be stored in a persistent and recoverable manner. The database is also updated with important run time parameters like the recently detected changes, for notifying the users. In case the system crashes, for the system to recover to a stable state and start serving the sentinels which were alive, data needs to be retrieved from the database. For example: change detection module needs the following information 1) URL to be monitored, 2) the change and compare specifications, and 3) the start and end of a sentinel. But the notification module requires 1) contact information and 2) notification mechanism to notify the changes. User information, such as 1) sentinel installation date, 2) the versions of a page for change detection and 3) storage path of detected changes are also stored, to allow the user to keep track of their sentinels. Queries are posed to retrieve the required data from the tables at run time using a JDBC bridge to display these information in the DashBoard.

3.4 Change Detection Module

WebVigiL uses a change detection module to perform change detection at a very fine level of granularity. The various components of change detection are discussed below.

3.4.1 ECA Rule Generator

In WebVigiL, every valid request that arrives triggers a series of operations that occur at different points of time [23, 24]. Some of the operations are: creation of a sentinel (based on start time), monitoring the requested page, detecting changes of interest, notifying the user(s) of the change, and deactivation of sentinel. In WebVigiL, for every sentinel, the ECA [25, 26] rule generation module generates event-condition-action (ECA) rules to perform some of the above mentioned operations. Briefly, an event-condition-action rule has three components: an event (occurrence of an event), a condition (checked when the associated event occurs), and an action (operations to be carried out when the condition evaluates to true). The ECA rules [27, 28] along with the local event detector(LED) [23, 24] are used:

1. To generate fetch rules for fetching required pages at specified time points.
2. To detect events of interest and propagate pages to detect primitive and composite changes.
3. To perform activation and deactivation of sentinels.
4. To make time-based notification of changes.

Activation/Deactivation: WebVigiL accepts interval-based monitoring requests, which means that the request has a start point and an end point within which the monitoring has to be performed. Each sentinel is enabled between the starting and the ending points. A sentinel can be disabled (does not detect changes during that period) or enabled (detects changes) by the user during its lifespan. In WebVigiL, the ECA rule generation module creates appropriate events and rules to enable/disable sentinels. As WebVigiL also supports sentinels defined on previous sentinels, ECA rules are an elegant mechanism for supporting asynchronous executions based on events.

Generation of Fetch Events: Periodic events [25] are defined and rules are associated for fetching with the frequency of fetch specified by the user. Whenever a

periodic event occurs, the corresponding rule is fired, which then checks (condition part of the rule) for change in meta-data of the page and fetches the page (action part of the rule) if there is a change in meta-data. Thus a periodic event controls both the polling interval and the lifespan of the fetch process. A fetch rule is created and used to poll the page of interest specified in the sentinel.

Generation of Notification Events: There are several options for delivering the notifications. One of the options is interval-based notification [6] in which the users specify the desired frequency for notification of changes. Again, we use ECA rules to perform the above. The system registers with the LED and creates periodic events [25] for each sentinel that requests interval-based notification. The event fires at regular intervals corresponding to the frequency specified by the users. When a periodic event occurs, the condition part of the rule is checked for any change that is detected since the last notification and notifies the corresponding user (action part of the rule) if a new change occurs. A notification rule is created to send time-based notifications to users.

3.4.2 Change Detection Graph

When two or more sentinels are interested on a particular change type on the same page, the redundant computation of change should be avoided. This problem is addressed by grouping the sentinels interested on the same change type, on the same page. The relationship between the sentinels and page is captured using a graph [4], called the change detection graph. The graph aids the change detection module in being less computationally expensive by maintaining certain runtime information. For example consider sentinel s1 on the URL "www.cnn.com" interested in the change types "Images AND Links" and sentinel s3 interested on the same URL on the change type "Images". The graph constructed for s1 and s3 is as shown in Figure 3.2.

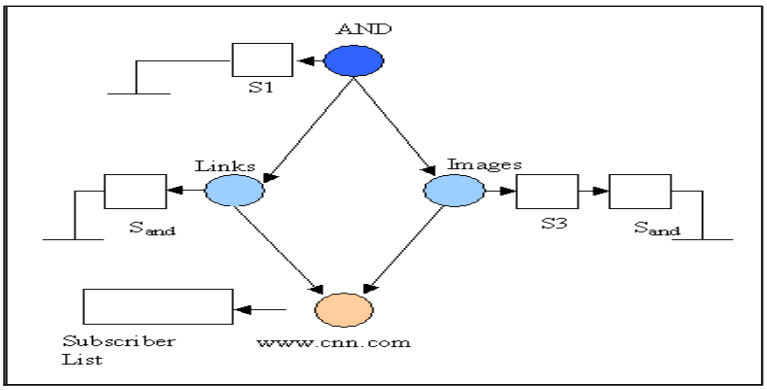

Figure 3.2 Change Detection Graph

URL node: The leaf node in the graph that denotes the page of interest is termed as the URL node (www.cnn.com). Hence the number of URL nodes in the system will always be the same as the number of distinct pages referenced by sentinels in the system at any point of time.

Change type node: All the nodes at level one in the graph are change type nodes (images, links). These nodes represent the type of change that is specified on a page. The change type nodes supported by the system are: all words, links, images, keywords, phrases, table, list, regular expression, and any change.

Composite Node: A Composite node is any node above level 1, which represents a combination of change types. For example "all links and all images". Currently, the system supports composite changes on a single page.

As seen in the graph, the relationship between nodes at each level is captured using subscription/notification mechanism. All the higher level nodes subscribe to the lower level nodes. The lower level nodes maintain the subscription information in their subscriber's list. The list corresponding to URL node consists of all the change type nodes that are subscribed to it. At the change type nodes each sentinel will have a subscriber that will contain the references to the composite nodes. Thus when a page is fetched,

the corresponding URL node is notified and the information is propagated to the higher level nodes. The change is computed at the change type node between the most recently fetched page and the reference page that is selected according to the compare option specified for that sentinel (as discussed earlier). If the page changes, the sentinels interested on the change (the sentinels in the subscriber list) are notified. The change type node being a part of the composite node, the later is also notified of the change. In the above graph change of images is computed only once for s1 and s3. As s3 is interested only in images, it is notified as soon as the change occurs. But s1 is interested in a composite change (images and links), so a change is propagated to the composite node (AND) by the change type nodes "images" and "links", only then s1 is notified.

3.4.3 Change Detection (CH-Diff & CX-Diff)

Currently, change detection has been addressed for Hypertext Markup Language (HTML) and eXtensible Markup Language (XML), the two most popular forms for publishing data on the web. Change detection algorithms [21, 5] work according to the change type specified by the users with the input. The corresponding objects are extracted from the versions of pages being compared. The versions to be compared for change detection are decided by the option which is chosen during the input. The extracted objects are compared for changes and if there are any, they are reported to the notification module for the corresponding user to be notified. In HTML, changes are detected to content-based tags such as links and images, presentation tags and changes to specific content such as keywords, phrases etc. As XML consists of tags that define the content, currently, we support change detection only to the content.

3.5 Fetch Module

The fetch module fetches the pages of interest that are given by different users according to their interests. The fetch module fetches the pages based on the specified frequency for each page. The options for specifying the frequency are: user-defined frequency (for e.g., every 2 days) or system-defined. This module thus serves as a local wrapper for the task of fetching pages. This module informs the version controller of every version that it fetches, stores it in the page repository and notifies the CDG of a successful fetch. The properties of a page are checked for determining the freshness of a page. The two properties being used are: Last Modified Time (LMT) or size of the page (Checksum). Depending upon the nature (static/dynamic) of the page being monitored, the complete set or subset of the meta-data is used to evaluate the change. Only if a change is detected using the properties, the wrapper fetches the page and sends the page to the version controller for storage. Fetch rules are created and used to poll the page of interest specified in the sentinel with the frequency specified as the interval of polling.

Best Effort Rule: In situations where the user has no information about the change frequency of a page, it is necessary to tune the frequency with which the page is fetched to the actual frequency with which the page is changing. BE Rule uses a best-effort Algorithm (BEA) [29] to achieve this tuning. In the best-effort algorithm, the next fetch interval (Pnext) is determined from the history of observed changes on that page. When the next polling interval is determined, the BE Rule changes the interval "t" of the periodic event.

Fixed-Interval(FI) Rule: In this case, the user explicitly provides a fetch frequency to fetch the required page. A periodic event [25, 26] with periodicity (interval t) equal to the user-specified interval is created and an FI rule is associated with it to fetch the page. As a result there will be more than one FI rule on a given page with different or same periodicity, where each rule is associated with a unique periodic event (i.e., with

different start and end times). The current model of fetching web pages for fixed-interval sentinels results in redundancy.

3.6 Version Management Module

There can be situations where more than one user may be interested in the same web page. In such cases the system has to connect to web page multiple times for the same version of the page. A repository service like version management module, reduces the number of network connections to the remote web server by avoiding redundant fetches, thereby reducing network traffic. The repository also provides the change detection module with two versions of a page of interest for detecting changes. The requirement of the repository service is, if the same version of a page is required twice for two different requests, the page should not be fetched twice but a stored version of the page should be used. The repository service should also be able to manage multiple versions of pages without excessive storage overhead. So, there is a necessity to store only those versions that are needed by the system and to purge all the versions which are not useful. The version management module also has a deletion algorithm [6] which considers the specifications of the sentinels requesting a particular page to identify the set of versions that are required for the system and deletes the remaining versions.

3.7 Presentation & Notification Module

The functionality of this module is to notify the users of detected changes and present the changes in an elegant manner. The computed differences should be presented or displayed in an intuitive and meaningful way, so that the user can easily interpret the changes. Two schemes are presented for presentation, namely, only change approach and the dual frame approach. In only change approach, the changes that have been made to

a page are displayed in a tabular manner where as in the dual frame approach, the the old and new versions of the page are shown side by side, with the changes highlighted. This thesis provides some extensions and changes to the presentation of the HTML and XML pages. It also discusses about the DashBoard which will serve as the user interface for WebVigiL. Chapter 6 discusses these extensions in detail.

The system provides users with a choice of the medium of notification and the frequency with which the detected changes has to be notified. The notification module delivers notifications to the users with a frequency given during the installation of sentinel. Currently the medium of notification is email only. The functionality of WebVigiL also provides the users with a DashBoard where the users can go and look up the changes and manage their sentinels. The WebVigiL server, based on the notification frequency can push the information to the user, thus implementing the "just in time"(JIT) paradigm.

Current WebVigiL system runs as an application looking for inputs in a specified port number. The web interface, made using Java Server Pages (JSP), receives and validates the inputs from the users and forwards the same to the WebVigiL server. The sentinel is then validated at the server side and informs the client of any invalid specifications. Once the sentinel is submitted, it is persisted using the KnowledgeBase module. The sentinel is then sent to change detection module to generate ECA rules according to the specification. The pages are fetched using the fetch module when the rules get activated. The fetched pages are forwarded to the respective change detection modules to detect changes. Once the changes are detected, it is time for notification module to notify the users of the change conforming to the specification.

CHAPTER 4

ADAPTIVE LOAD BALANCING FOR WEBVIGIL

WebVigiL is envisioned to be web-based service that is expected to be used by a large user community. Hence, it is imperative that it provides $24 * 7$ service. Unlike search engines, a sentinel needs to be serviced over a period of time (i.e., lifespan of the sentinel). As a consequence, WebVigiL also needs the ability to scale to handle large volumes of client requests without creating delays. Having only a single WebVigiL server is insufficient to handle the amount of traffic, or load, WebVigiL receives. A load balancer can be used to increase the capacity of WebVigiL beyond that of a single server. It can also allow the service to continue even in the face of server down time due to server failure or server maintenance.

In WebVigiL, there can be situations where the server might get loaded with requests, which in turn can lead to situations where a large number of pages need to be fetched at the same time, detect changes between pages and notifying them. This can become a bottleneck even though fetching and change detection are executed as seperate threads. In spite of change notification being a separate thread, There may be delays in notification due to a large number of requests and all the threads have to time-share a small number of processors. This may result in delayed responses and may even lead to break down of the server due to limitations of Memory, Disk space and processing capacity of the server. Hence, a load balancing mechanism is required to distribute client requests intelligently to a WebVigiL server from the pool of available servers.

The traditional methods used for load balancing server farms are: least connections, round robin, least response time, least bandwidth and least packets. Most of these

are used in routing where the semantics of the task is simple and same for all requests. Since we are developing load balancing methods which need to adapt to the semantics of sentinels and the architecture of WebVigiL, these methods are not be suitable. For WebVigiL, the requirement of Load Balancing is to distribute the sentinels among the several servers so that the change detection can be scaled by utilizing the existing architecture. The above mentioned methods are not yet fine tuned to be ably applied in their original form.

This chapter discusses various issues of adaptive load balancing for WebVigiL. They include identifying load factors, estimating the load factors, distributing the load to multiple WebVigiL servers and adapting to the changing load conditions of WebVigiL servers.

4.1 Load Factors of WebVigiL

Every sentinel in WebVigiL adds different types of load to a server. Apart from the processing of a sentinel and setting things up, three types of load exerted by a sentinel is of significance: i) the number of fetches added by a sentinel over its lifespan and the time at which it needs to be done. The fetch involves network delays (URL may be local or international) and the amount of data fetched also adds to the load, ii) after the page is fetched, they need to be stored and managed. Depending upon the lifespan and frequency, there is a load on the secondary storage, and iii) change detection on two pages can take considerable amount of time as it is mostly string processing. We consider the above three as major load contributors for each sentinel submitted to the WebVigiL system.

4.1.1 Fetching

After the client sends a request to the WebVigiL server, it creates events and rules for fetching the page. As the number of requests increases there will be be cases in which there will be multiple sentinels on a same page with same or almost the same fetch intervals. WebVigiL system uses the concept of grouping [7] sentinels based on URL and fetch frequency to reduce the number of fetches. WebVigiL also reduces the number of fetches by fetching the page only when the LMT or checksum of the page changes. But When the number of sentinels with different URLs set on the same time increases, the group of sentinels becomes larger, so one sentinel has to wait for another sentinel to finish its fetching. This introduces delay which may be propagated into the notification of the change detection as well. For example, consider the following scenario. Assume there are 1000 sentinels set to be started at the same time (e.g. 10:00 AM). WebVigiL cannot fetch all the sentinels exactly at 10:00AM. If fetching one sentinel takes 1 second, it takes 17 minutes to fetch all the sentinels. But if we can fetch 2 sentinels every second it will take 8 minutes 30 seconds.

4.1.2 Version Management

Since change detection requires two versions of the same page for comparison and different versions of the same page are retrieved over the lifespan of a sentinel, there is a need for storing the pages. The stored pages are also supplied to various modules as required. In addition, the web pages created for presenting the changes also grows considerably. WebVigiL system uses an in-built deletion algorithm and a DIFF [8] based approach to reduce the number of versions and the amount of data per page stored in the disk, respectively. The amount of pages stored can also grow considerably when the lifetime and frequency of sentinel is very large. If the amount of disk space is limited, WebVigiL server becomes handicapped and will not be able to perform as required. For

example consider the following scenario: If average size of a webpage is about 60 kb (based on out experiments) and is fetched every second for a day, it adds up to at least 5 GB of disk space

4.1.3 Change Detection

Change detection takes place for both HTML and XML pages. After the page is fetched using one of the fetching methods, the sentinel is propagated to the change detection module which uses the appropriate algorithm (CH-DIFF or CX-DIFF). The change detection mechanism extracts objects from the pages to detect changes. Time taken to parse the page, extract the objects and compare them to detect changes is related to the amount of content present in the page which is based on the size of the page. This can be considered as a load factor because if more of these type of sentinels, which have large page size are requested by the clients, then the time taken to process these sentinels one by one will lead to a ripple effect of delay in change detection for subsequent sentinels.

4.2 Estimating Load

In WebVigiL, sentinel adds load to the system. Whatever the load, the system may come under is due to the sentinels. So it is possible to estimate the load using the sentinels.

4.2.1 Estimation of Fetch Load

To estimate the Number of fetches a system can handle, it is required to know the number fetches required for a single sentinel. For example consider a sentinel S1 with the following requirements.

Sentinel S1 on http://www.cse.uta.edu

Monitor LINKS

Fetch every 1 hours

From: Now To: Now + 10 days

Compare Pairwise

Since the above sentinel is a Fixed Interval sentinel, the system tries to fetch the page every hour. The total number of fetches can be calculated using equation 4.1.

$$n = [l/f] \tag{4.1}$$

WHERE

- f = Frequency at which the page is fetched (fetch/minutes)
- l = Lifetime of the sentinel (minutes)
- n = Number of fetches

For the above example, the number of fetches n can be calculated as 240 because the frequency f is for every hour and the lifetime l is for 10 days. If the monitoring URL is a dynamic web page, the page is generated dynamically and does not have a LMT. Even the checksum of the page varies every time the page is fetched. So the worst case scenario of 240 fetches is observed. Now, conside a static web page, the LMT and Checksum does not change until the page changes. Since the LMT and Checksum does not change, the web page will not be fetched. As there is no way to predict the change frequency of a web page, the exact number of fetches cannot be determined. It is reasonable to over estimate the number of fetches a sentinel may need by estimating the worst case scenario.

Now consider k sentinels installed in the system. If all the sentinels point to unique URL's then the fetch load can be calculated using the equation 4.2.

$$\begin{aligned} n &= [n_1 + n_2 + ... + n_k] \qquad n_i = [l_i/f_i] \\ n &= \sum_{i=1}^{k-1} [l_i/f_i] + [l_k/f_k] \end{aligned} \tag{4.2}$$

If the sentinels are dependent, WebVigiL server does grouping to reduce the number of fetches. Therefore while estimating the fetch load the equation has to take into account of the same. If the above said k sentinels were dependent then grouping of these sentinels would form g groups. After grouping the fetch load can be calculated using the equation 4.3.

$$n_k^g = [n_1 + n_2 + ... + n_g] \qquad n_i = [l_i / f_i] \tag{4.3}$$

WHERE

- f_i = Maximum frequency of the sentinel in a group (fetch/minutes)
- l_i = Maximum lifetime of the sentinel in a group (minutes)
- n_k^g = Number of fetches for g groups

All the incoming sentinels do not add to the fetch load. This is because if more than one sentinel belong to a group then the sentinel which has the maximum frequency is used for fetching pages for that group. Therefore an incoming sentinel adds to the fetch load if it falls under the following criteria.

- If the sentinel does not belong to any group
- If it belongs to a group and has high fetch frequency ($f_{k+1} > f_g$)

Therefore equation 4.3 can be modified as follows to suit the above conditions.

$$n_{k+1}^{g+1} = [n_1 + n_2 + ... + n_g + n_{g+1}] \tag{4.4}$$

Equation 4.4 represents the situation where the incoming sentinel may be of unique URL or an existing URL with different fetch frequency. Therefore on grouping it adds itself as a new group, thus contributing to the fetch load.

$$n_{k+1}^{g} = [n_1 + n_2 + ... + n_{k+1}] \tag{4.5}$$

Equation 4.5 represents the situation where the incoming sentinel may belong to an existing URL with maximum fetch frequency for a group. In the above equation $f_{k+1} >$

f_g, therefore the term n_g is replaced by n_{k+1}. The above estimate is pretty much a safe estimate which can be used for determining the number of fetches when new sentinels are added to the system.

4.2.2 Estimation of Disk Load

To estimate the amount of disk space used by WebVigiL system, it is required to know amount of disk space used by a single sentinel. The disk space used by a single sentinel can be calculated using the average size of the page and the number of fetches that sentinel performs. Therefore the disk space required is given by the equation 4.6

$$d = [a * n] \tag{4.6}$$

where

- a = Average size of the page (kilobytes)
- n = Number of fetches (fetches)
- d = Disk space usage (kilobytes)

Assuming the average page size a is about $60Kb$, using the number of fetches n as 240 from the previous example (sentinel *S1*), the disk space d required for that sentinel can be calculated to be about $14400Kb$. In order to calculate the disk space usage for a number of sentinels, we consider k sentinels which are not overlapping and specified on unique URL's. Then we can calculate the total disk space usage d using the equation 4.7

$$d = [d_1 + d_2 + ... + d_k] \tag{4.7}$$

Now consider overlapping of sentinels and having more than one sentinel installed on the same URL. If more than one sentinel is pointing to the same URL, then sentinels are grouped based on their fetch frequency to reduce the number of fetches and all the fetched pages for a URL is stored at the same folder. Therefore the equation has to be

altered to represent the scenario. If we have k dependent sentinels then they can be grouped in to g groups. Therefore the disk space used by g groups can be calculated using the equation 4.8.

$$d_g^k = [d_1 + d_2 + ... + d_g] \qquad d_i = [a_i * n_i]$$
$$d = \sum_{i=1}^{g} [a_i * n_i] \tag{4.8}$$

The above equation was formed assuming each and every fetched page is stored in the disk. But, for each URL there will be only certain number of versions stored. That number of versions is determined by an inbuilt deletion algorithm. The deletion algorithm uses the following sentinel properties to calculate the number of versions required by all the sentinels belonging to an URL group.

- d (Deletion count) = The number of versions required depending upon the compare option (Pairwise, Moving n, Every n)
- h (Max Change History) = The number of versions associated with a page regardless of the sentinel

When deletion is enabled in the WebVigiL server, only a certain number of versions will be stored in the disk. But the above equation does not represent the scenario; it needs to be adjusted to estimate the disk space correctly. Therefore we modify the equation to estimate the amount of disk space required when deletion is enabled.

$$d = \sum_{i=1}^{k} [a_i * (F_i / f_i) * c_i * h_i] \tag{4.9}$$

where

- a_i = Average page size of that URL (kilobytes)
- c_i = Deletion count [2 (Pairwise), n (Every, Moving)]
- h_i = Maximum change history [3 (default)]
- F_i = Minimum fetch frequency for that URL

- f_i = Maximum fetch frequency for that URL

The ratio of Maximum and Minimum frequency in equation 4.9 gives the number of versions common between the sentinel of low frequency and sentinel of high frequency. When this ratio is multiplied with the deletion count it gives the total number of versions required for that group. The change history factor is again multiplied to give a maximum limit on number of versions stored in the disk for that URL. Thus, recursing through all the URL will give the amount of disk space required for all sentinels. After considering all the factors involved in storage of versions we can safely estimate the amount of disk space required when more sentinels are added to the system.

4.2.3 Estimation of time taken for Change Detection

Change detection takes place once there are two versions of a page. The change detection graph propagates the versions to the intended change detection node after parsing the objects from both the pages. Then the change detection node uses the two objects to determine the changes. Major load that can be observed in the change detection module is the time taken to detect changes and includes time for extracting the objects and comparing them. In order to estimate the load exerted on this module by all the sentinels, it is required to know the amount of load exerted by a single sentinel. To calculate the time taken for change detection t for one sentinel we need the average page size a and the time taken to change detect page of unit size $($u). Therefore an equation can be derived to show the same.

$$t = [a * u] \tag{4.10}$$

Change detection is activated whenever a fetch takes place. Therefore there is a need to insert the number of fetch term n to accurately calculate the time taken for change detection for a sentinel during its lifetime, which is shown in equation 4.11

$$t = [a * u * n] \tag{4.11}$$

WebVigiL has more than one sentinel installed at a given time. If we consider all these k sentinels points to unique URL's then we can calculate the amount of time taken to detect changes as sum of time taken to change detect each and every URL, which can be showed by equation 4.12.

$$\begin{gathered} t_k = [t_1 + t_2 + ... + t_k] \qquad t_i = [a_i * u * n_i] \\ t = \sum_{i=1}^{k} [a_i * u * n_i] \end{gathered} \tag{4.12}$$

Now consider the scenario when all these k sentinels do not point to unique URL's. i.e, when all the URL's in the system and incoming sentinels can be grouped based on their URL and fetch frequency. On grouping they form g groups. Now the time taken to change detect can be shown by the equation 4.13.

$$t_k^g = [t_1 + t_2 + ... + t_g] \tag{4.13}$$

If the load balancer is able to estimate the amount of time taken to detect changes to each sentinel and limit the number of sentinels added to the system, it is possible to detect changes for all the sentinels without much of a delay. WebVigiL reduces the number of fetches but it does not reduce the number of comparisons (unless both sentinels are interested in the same changes). Because it is necessary to detect smallest of changes.

When A sentinel is submitted to WebVigiL, it is received by the load balancer; it tries to fetch the URL and find the size of the page. Once the size of the page is known, it is possible to know the amount of time it takes to detect change. If the load balancer

adds the amount of time taken for each and every sentinel, it will be able to limit the number of sentinels if capacity for that server is reached.

4.3 Distributing the Load

The Load Balancer has to distribute the load among a number of servers to actually balance the load on each server. Instead of following a random strategy or other traditionally available strategy, we will explore strategies that make use of the semantics of sentinels and hence will be more appropriate than using a general strategy. We propose the following strategies for WebVigiL: 1) Maximize Each server strategy 2) Round Robin strategy and 3) Attribute based strategies. Below, we will discuss each of these strategies in detail.

4.3.1 Maximize Each Server strategy

In this strategy one server is used to its maximum capacity before another server is used. This strategy is similar to a situation where there is only one WebVigiL server. This strategy is designed such that when there is not much of a load on the server, it is possible to run this strategy so that resources are not wasted. Since the load is added incrementally to only one server the various estimates are calculated to prevent the server from being overloaded.

Sentinels add to the load of the WebVigiL system. This load is estimated using the formulas discussed earlier and stored in the data structures of the Load Balancer for further use. The server is used till its capacity is reached. When the capacity of the server is reached another server is started by the load balancer and the incoming sentinels are added to the second server. There might be cases when the first server becomes available, as sentinels expire and the load is reduced. Since the load balancer frequently updates

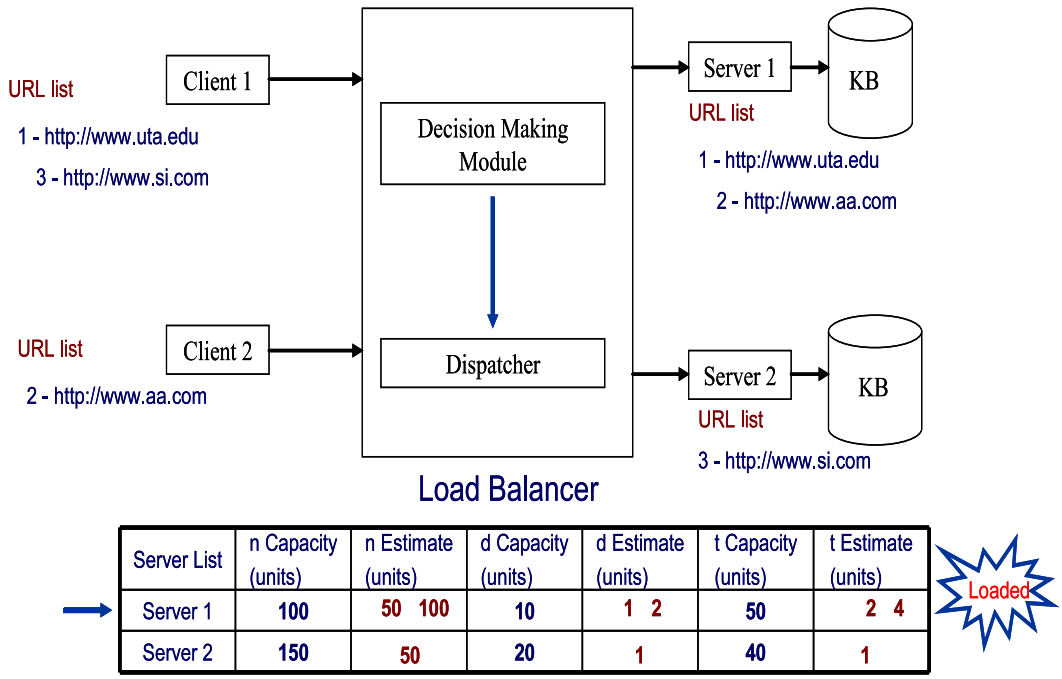

Figure 4.1 Maximize Each Server strategy

itself with the actual load from each server, it is possible for the Load Balancer to detect that and maximize the server again to its capacity.

Figure 4.1 shows pictorial representation of Maximize Each server strategy. Initially client 1 sends a sentinel to the Load Balancer. Maximize Each strategy chooses a server for an incoming sentinel. Decision maker then calculates the estimate load for that sentinel and stores it in the internal data structure. The decision maker passes the server and the sentinel object to dispatcher for installing the sentinel on the chosen server. Once the communication with the server is done, the sentinel is installed in the server for further processing the change detection request. But server 1 gets loaded when sentinel from client 2 is installed. Therefore when a new sentinel request from client 2 comes to Load balancer it is assigned to server 2 for further processing. Maximize Each server continues distributing the sentinels in this manner.

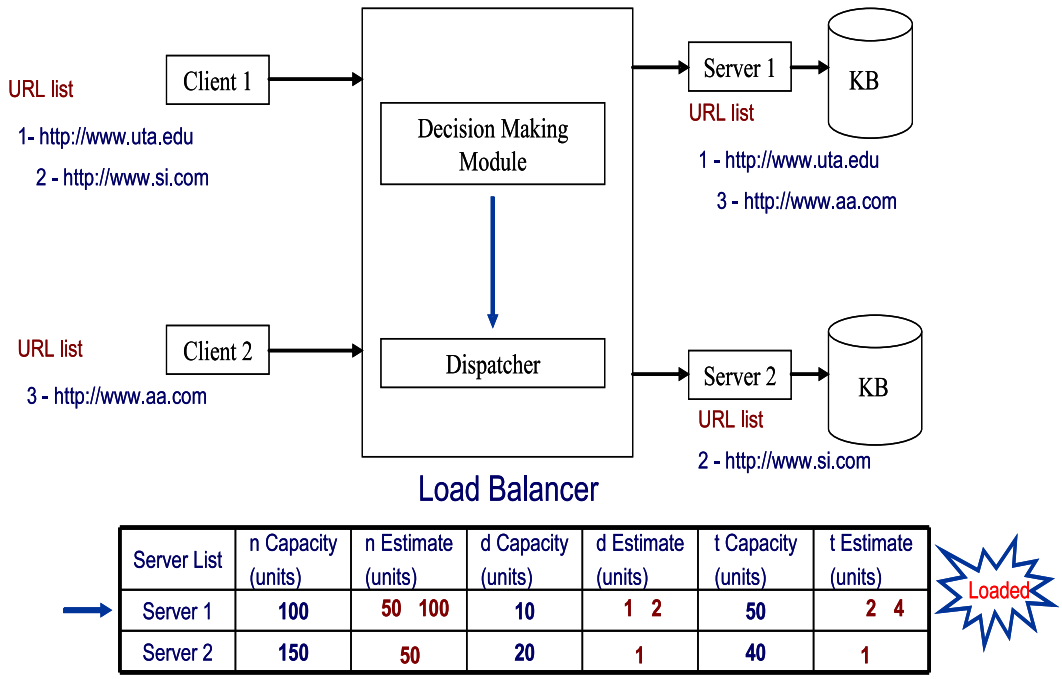

Figure 4.2 Round-Robin strategy

4.3.2 Round Robin strategy

In this strategy sentinels is distributed among a group of servers. This strategy must at least have 2 servers to start with. Since this strategy is trying to improve the performance over the first strategy, it is necessary to have more resources under its disposal. Since the main aim of the strategy is to reduce the load on WebVigiL system, performance will be certainly better when the same number of sentinels are distributed among two servers using this strategy.

Figure 4.2 shows pictorial representation of Round Robin strategy. Initially client 1 sends a sentinel to the Load Balancer. Round Robin strategy chooses a server for a incoming sentinel. Decision maker then calculates the estimate load for that sentinel and stores it in the internal data structure. The decision maker passes the server and the sentinel object to dispatcher for installing the sentinel on the chosen server. Once the communication with the server is done, the sentinel is installed in the server for

further processing the change detection request. When a new sentinel request from client 1 comes to Load balancer it is assigned to server 2 for further processing because of the round robin strategy. Thus round robin continues distributing the sentinels. The biggest disadvantage of this strategy is that similar sentinel when given in quick succession may not be grouped and server resources may be wasted in fetching the same page by two servers. This strategy also does not use the sentinel properties for installing the sentinels in servers.

4.3.3 Attribute-Based strategies

This uses sentinel properties to distribute sentinels to its destination servers. The various properties that can used are: 1) URL of the sentinel, 2) change type of the sentinel, and 3) fetch frequency of the sentinel. This strategy is specifically designed so that it makes efficient use of WebVigiL server's implementation. When sentinels belonging to the same URL are grouped in a server, the server will do less fetching. If sentinels were separated based on whether it is a Fixed-Interval (FI) or Best-Effort (BE) sentinel then it significantly reduces the version lists stored by WebVigiL server to maintain versions for both FI and BE sentinels.

Figure 4.3 shows pictorial representation of Attribute-based strategy. Initially client 1 sends a Fixed Interval (FI) sentinel to the Load Balancer. Attribute based strategy chooses a server for an incoming sentinel. Decision maker then calculates the estimate load for that sentinel and stores it in the internal data structure. The decision maker passes the server and the sentinel object to dispatcher for installing the sentinel on the chosen server. Once the communication with the server is done, the sentinel is installed in the server for further processing the change detection request. When a new Best Effort (BE) sentinel comes from client 1 to Load balancer it is assigned to server 2 for further processing because of the attribute based technique.

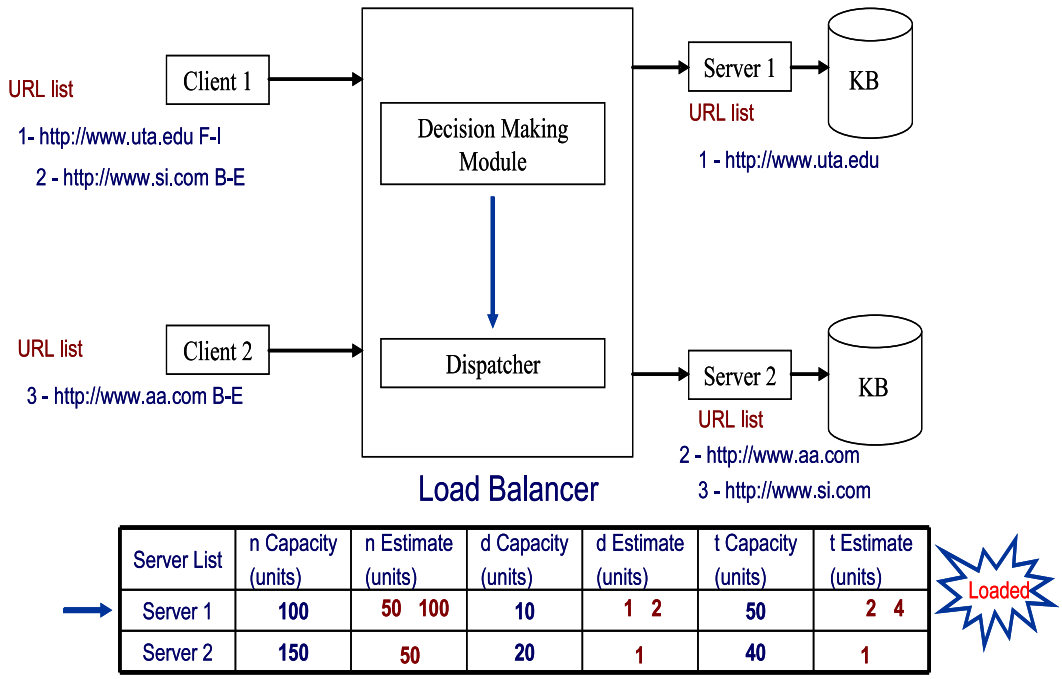

Figure 4.3 Attribute based strategy

4.4 Adapting to the Load

The equations discussed in previous sections estimate the load of the sentinels when they are added to the system. Since the sentinels have a deterministic lifetime, the load exerted on a sentinel may reduce considerably when sentinels expire. But the load balancer still continue to have old details about the server and will not be able to adapt to changing conditions of the server. Therefore a feedback mechanism is necessary to update the load balancer about actual load conditions in the server. The load balancer has to frequently retrieve information from the WebVigiL servers to get the actual load factor and update its estimate so that load balancer can continue to distribute the sentinels efficiently based on fresh estimates. Since the estimated capacity and actual values needs to be used by load balancer for both decision making and adapting, a shared data structure is used. If a server is loaded it is represented by a boolean variable in the shared data structure. On updating the shared data structure, if we find the server is not loaded

then the Boolean variable has to be reset to enable addition of more sentinels to a server. Even the data structure which contains the grouping information has to be updated to remove sentinels whose lifetime has ended. Thus by frequently updating the estimates and groups, load balancer will be able to accurately estimate the load and dispatch the sentinels to the correct servers. This can also be termed as adapting to the changing load conditions.

4.5 Summary

This chapter discussed in detail how load factors are calculated or estimated for sentinels submitted to WebVigiL. These load factors were estimated using static formulas. Once the static formulas were established, it is easier to form strategies for distributing the load among a cluster of servers. Once a sentinel is distributed to a server, the load balancer has to update itself with actual values to continue load balancing in an efficient manner. In summary, load factors were identified, estimated, distributed, and adapted for WebVigiL servers.

CHAPTER 5

LOAD BALANCER ARCHITECTURE AND IMPLEMENTATION

Load Balancer is designed to follow content-switch method for balancing the load. That is, the load balancer should be able to intercept the incoming sentinels, check its contents and distribute them accordingly. The system comprises of various modules, which constitute the architecture. The rest of the section describes, briefly, how various alternatives were taken into consideration, to build a simple and effective load balancer for WebVigiL.

5.1 Analyzing WebVigiL architecture for load balancing

Figure 5.1 shows a typical Load Balancer used for increasing the capacity of a server farm beyond that of a single server. It can also allow the service to continue even in the face of server down time due to server failure or server maintenance. A load balancer acts as a virtual server having its own IP address and port. This virtual server is bound to a number of services running on the physical servers in a server farm. These services contain the physical server's IP address and port. A client sends a request to the virtual server, which in turn selects a physical server in the server farm and directs this request to the selected physical server. Load balancer's role is to manage connections between clients and servers and to distribute load using any one of the strategies.

Figure 5.2 shows system components of WebVigiL. The web-interface is written in JSP, so it uses the tomcat server. when a client uses the interface to give inputs to the server, the request is forwarded to WebVigiL server through a predefined port. WebVigiL

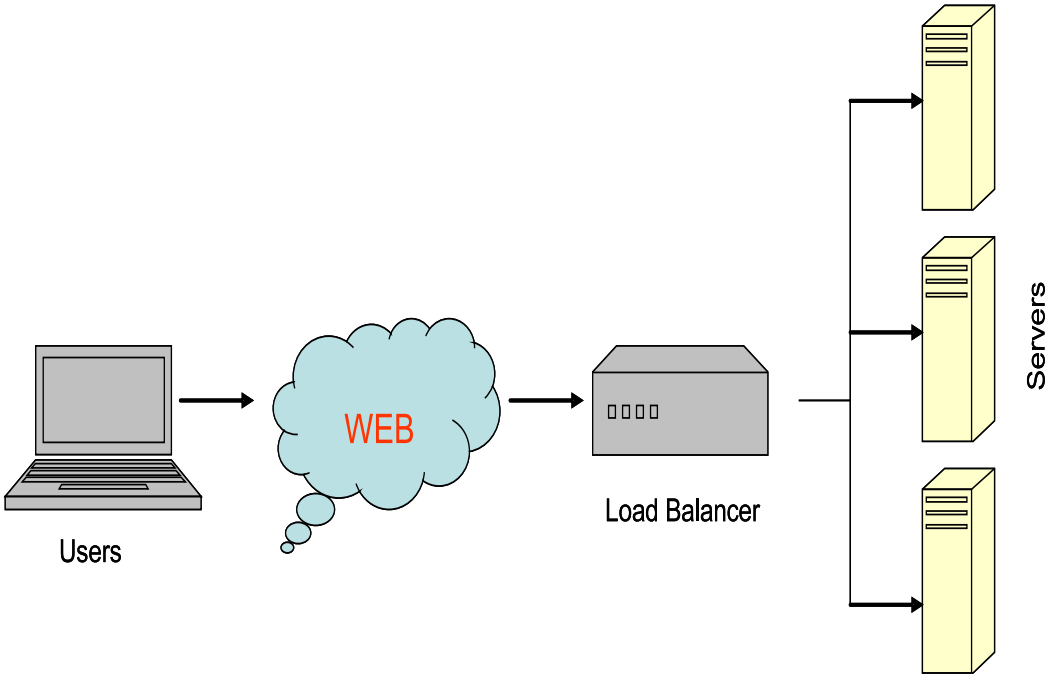

Figure 5.1 Typical Server Load Balancer

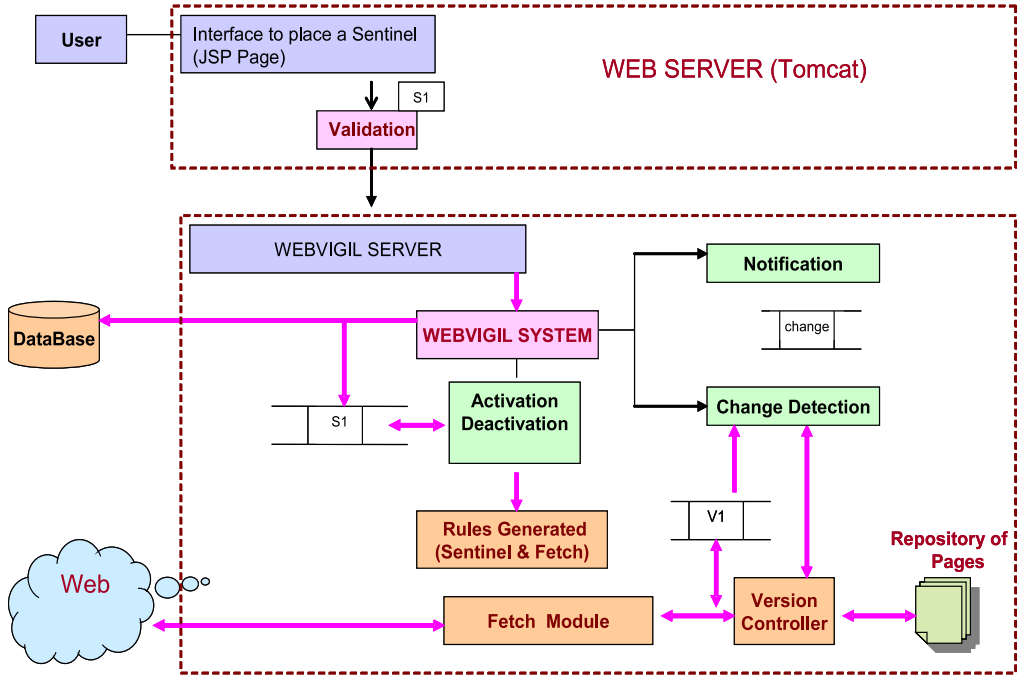

Figure 5.2 WebVigiL system components

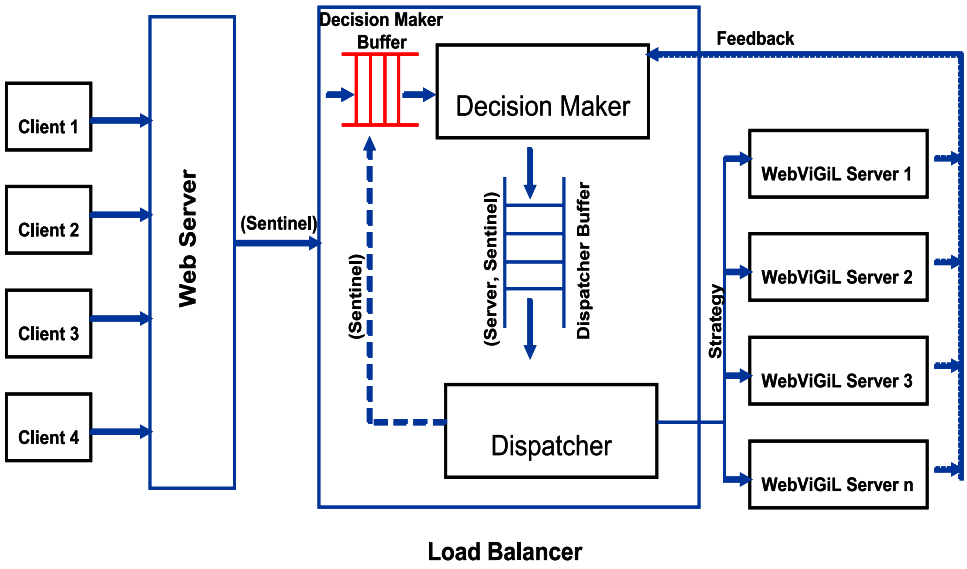

Figure 5.3 Load Balancer Architecture

server uses the web to fetch the page of interest to do change detection and for notifying the detected changes.

5.2 Load Balancer Architecture

In order to apply Load Balancing for WebVigiL, the architecture of a typical load balancer can be adopted. Load Balancer is located between the Web-server and WebVigiL server, which can be explained by the following reasons:

- Load Balancer balances load on WebVigiL servers.
- Load Balancer handles only inputs (sentinels) from clients.
- Load Balancer distributes load among WebVigiL servers.
- Load Balancer adapts to changing load conditions in WebVigiL servers.

Figure 5.3 shows an overview architecture of Load balancer for WebVigiL.

5.2.1 Decision making buffer

Decision making buffer was made part of the Load Balancer because it enables buffering of incoming sentinels. The decision maker picks up the sentinels for processing from this buffer. If the decision maker is busy making decision for a sentinel then all the incoming sentinels are buffered for the decision maker. This prevents waiting of clients for load balancer to make decisions for their sentinels. The buffer also serves as a intermediate place for sentinels which haven't been sent to their allocated server because of server failure or communication failure.

5.2.2 Dispatcher buffer

Dispatcher buffer buffers sentinels for the dispatcher module. The decision making module makes decisions for the sentinel and adds the server and sentinel details to the dispatcher buffer. The buffer is there so that when dispatcher module is sending requests to servers for installing the sentinel, it may not be able to serve other sentinels. Since decision maker should not wait till dispatcher finishes its task, it can add to the dispatcher buffer.

5.2.3 Decision maker module

Decision maker module is the heart of the load balancer. This module is responsible for choosing the server for a given sentinel and maintains the load estimate information for all the servers. The decision maker module waits for a sentinel to be added to the decision maker buffer. When it retrieves a sentinel from the buffer, sentinel information is used to estimate the load this sentinel will exert on a given WebVigiL server. The calculated load factors are added to the existing estimated load for a chosen server to give total load on that server. This estimate is compared with the capacity of the server to determine whether the server has been used to its maximum capacity. In that case,

a boolean variable is set to determine if the server is loaded. If not, then the server will be used for another incoming sentinel. Now decision maker has made a decision on a sentinel and is ready to transfer the server and sentinel information to the dispatcher buffer.

5.2.4 Dispatcher module

Dispatcher module reads the server and sentinel information from the dispatcher buffer whenever it is available. The dispatcher then tries to connect to the chosen server. If the server is available then the given sentinel is installed in the server. It might be possible that the server is up and communication between the load balancer and the server may be down. The dispatcher can try repeatedly in a loop to connect to the server to install the sentinel. But since the load balancer should be able to handle large number of sentinels at once, it cannot use all its resources handling only one sentinel which is not able to reach a destined server. Therefore an intelligent decision for load balancer would be to send the sentinel information back to the decision maker to choose another available server for the sentinel. Using this technique might skew the estimates for servers in the load balancer for which a feedback module was designed to overcome this shortcoming.

5.2.5 Feedback

The load balancer has to adapt to the changing load conditions. But if the load balancer uses outdated load estimates to distribute sentinels among servers, then efficiency of the load balancer might reduce considerably in the long run. Therefore, there is a need for a module which updates the load balancer with the `actual`load values so that the load balancer's efficiency remains consistent. The runtime information of WebVigiL servers, such as: 1) Number of sentinels running, 2) Number of fetches, 3) The amount of disk space used, and 4) The time taken for change detection computations are stored in

a flat file by each server. The feedback module fetches these files from shared file space to update the load balancer's shared data structure. Once the data structure is updated the load balancer will be able to make better decisions in choosing a server. The grouping data structure also needs to be updated because the sentinels may end at anytime as specified by the user. Since load balancer cannot keep track of the end times of all the sentinels, a better approach is to update the grouping structure using the feedback information. The feedback module updates load balancer with actual load values at a configured time or when 80% of load is reached for a server.

5.3 Implementation of load balancer

The data structures required for load balancing and their implementation details are discussed below.

5.3.1 Storing and Updating load information

The runtime estimates of load are stored in a data structure for making decision on the next incoming sentinel. Figure 5.4 shows the data structure used for storing the load information. When the load balancer is started, all the WebVigiL servers load capacity information stored in a configuration file is used to update the data structure; this information is constantly used by the decision maker and updated by the feedback mechanism which necessitates the data structure to be designed as a shared data structure.

The configuration file contains the server name, the port number at which the server is listening, the user name and password for the machine, fetch load capacity, disk space capacity and the maximum time that can be taken for change detection. The values from the configuration file are read and updated accordingly for each server. Now when a new sentinel comes in, the estimates are calculated using the static formulas and the data structure is updated with the static estimate. When a subsequent sentinel comes in,

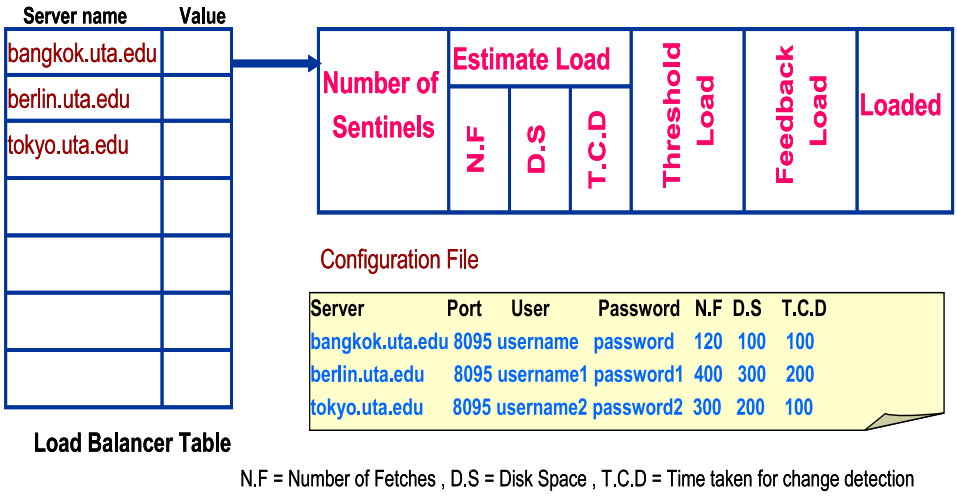

Figure 5.4 Data structure for storing load information

the previous estimate is used for incrementally computing load added to a server. After deciding to send a sentinel to a server, the estimate is compared with the capacity value for determining if the server is loaded or not. If the server becomes close to capacity or crosses the capacity the server is marked loaded. An incoming sentinel will not be assigned to this server until the server's load reduces. The reduced load information is obtained using the feedback mechanism which updates the load data structure frequently with the actual load information.

5.3.2 Grouping URL's based on fetch frequency

Sentinels placed on the same web page are grouped. All the sentinels that are placed on the same web page and having same fetch intervals can be grouped. Whenever the fetch intervals of two sentinels on the same web page differ by less than 5%, they are also be grouped [7]. Another case where sentinels can be grouped is when the fetch interval of one sentinel is a multiple of the fetch interval of another sentinel. For example,

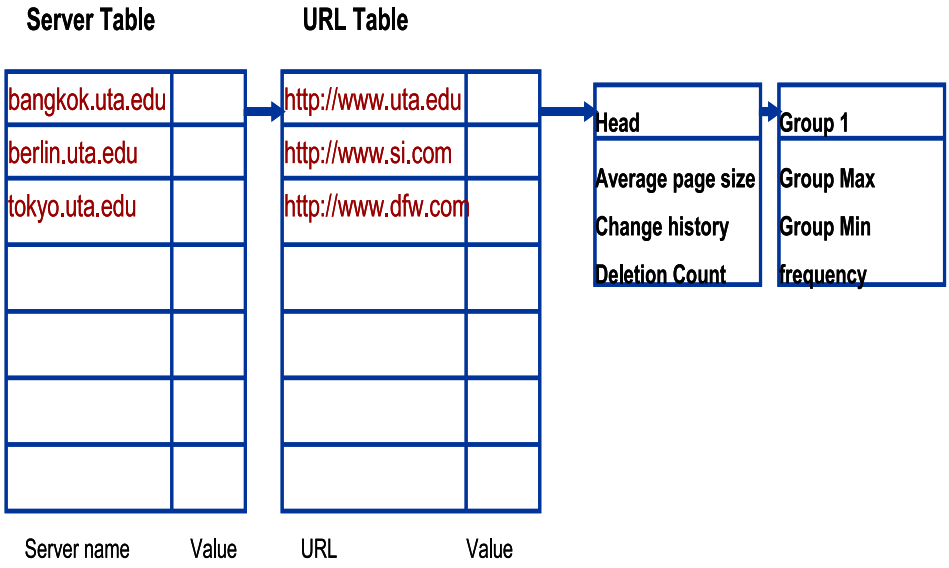

Figure 5.5 Grouping URL's based on fetch frequency

if a sentinel has a fetch interval of 1 hour while another sentinel has a fetch interval of 2 hours, they can be grouped. The multiplicity is determined at the minute level, meaning a sentinel with a fetch interval of 30 minutes can be grouped with a sentinel with a fetch interval of 1 hour. Sentinel groups are created based on the fetch intervals of various sentinels that can be grouped, but groups are also adjusted based on the addition of new sentinels or deletion of existing sentinels.

The number of groups on a given URL change based on the way new sentinels are added and old sentinels get deleted. To efficiently handle the process of addition and deletion of groups, different groups are maintained as a linked list. Every group in the list has a group identifier, groups maximum fetch frequency and groups minimum fetch frequency. The groups head contains the average page size of the URL, change history for that URL and the deletion count which is set based on types of comparison specified. Figure 5.5 shows the grouping data structure used in WebVigiL load balancer. This data structure also helps in estimating the load of the sentinels because the grouping data

structure contains the sentinels properties. Since sentinels have limited lifespan the data structure has to be updated frequently to have up to date estimate for distributing the sentinels efficiently. This is done by the feedback mechanism.

5.4 Summary

This section described various modules of the load balancer and its functions. It also described the implementation details of the load balancer. The various issues on storing and manipulating load estimates, grouping of sentinels and updating the data structure with actual load data were discussed.

CHAPTER 6

DASHBOARD AND IMPROVEMENTS TO CHANGE DETECTION

DashBoard is a set of utility tools provided as a user interface for users to track their inputs or sentinels in the system. Since WebVigiL is projected as a product, to detect changes on web pages, there is a need for a web-based interface which can be accessed by all users to place their monitoring requests and track them. This chapter discusses various factors that were considered in designing the user interface.

WebVigil facilitates users with two types of presentation, namely "Changes-Only Approach" and "Dual-Frame Approach". The users are notified by an email with the changes detected for their sentinels. The users can choose from two types of presentations mentioned above while installing a sentinel. If the changes are very frequent, the mailbox of the users will be flooded with email notifications. To avoid the same an option "interactive" notification is used while installing the sentinel. This chapter presents the improvements in the presentation technique of dual-frame approach.

6.1 User Interface

A web based user interface as shown in Figure 6.1 is provided to the users to submit their profiles termed as sentinels, disable/enable sentinels and view the detected changes. The user interface is made simple and easy for navigation. The user interface is implemented using HTML, CSS (Cascade Style Sheet), JSP (Jave Server Pages) and Javascript. Server side scripting (JSP) is used to process the input given by the users and direct it to the server for further processing. Every user that comes to WebVigiL web page can login into the system for placing a sentinel or retrieve information about their

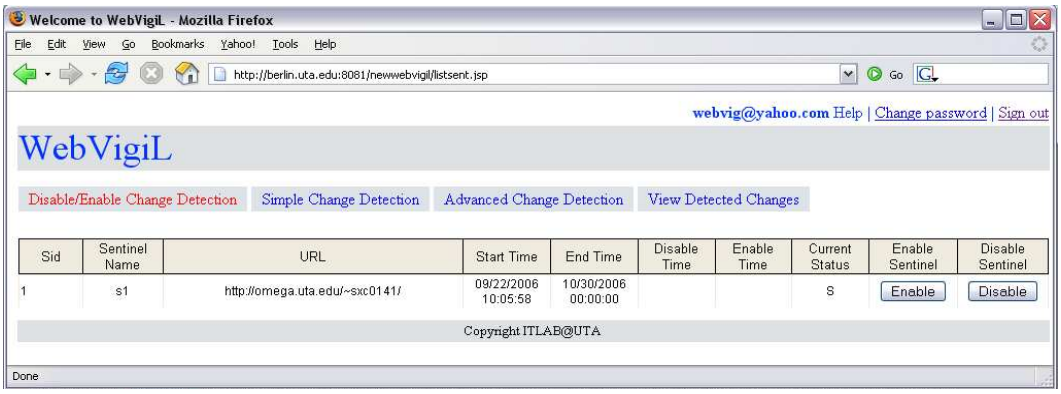

Figure 6.1 WebVigiL user-interface

previously defined sentinels. The interface allows the users to change their passwords and look for help if they need it. The interface has been divided into four parts, to make it flexible for the users to easily navigate through the interface. The four parts are as follows:

- Disable/Enable Change Detection
- Simple Change Detection
- Advanced Change Detection
- View Detected changes.

6.1.1 Disable/Enable Change Detection:

This section of the user interface allows the users to control their change detection requests. The interface displays all the sentinels set by the user. The list of sentinels has Disable and Enable button for each and every sentinel to control the behavior of the sentinel. The display of the sentinels is done using Data Grid Taglibs which has several features like sorting the headers and multiple pages of display. Only an active sentinel can be enabled or disabled. An expired sentinel cannot be disabled or enabled.

6.1.2 Simple Change Detection:

WebVigiL as an web service for change detection of web pages should be such that a naive user should be able to navigate, set monitoring requests and view the detected changes. With this is in mind, a simple change detection request interface was framed. This section allows a naive user to fill in his request for change detection in the form of 7 simple questions. The questions also have a default answer which the user can leave it unchanged. Various advanced features of WebVigiL have been made invisible to naive user. The interface is designed such that the user has to input minimal information as possible. Figure 6.2 shows a snapshot of simple change detection interface. The Simple Change Detection requests the following questions from a naive user:

- What naming they want to give their sentinel?
- What page they want to monitor?
- What Change type they are interested in?
- Where the notification is to be sent?
- How frequent the user wants the system to monitor the page?
- When the change detection has to start?
- When the change detection has to stop?

6.1.3 Advanced Change Detection:

WebVigiL has various features which are showcased in the advanced change detection section. This section has been designed to meet the requirements of a advanced user who wants to heavily customize the change detection request. Apart from the questions from the simple change detection, this section has few more questions which allows an advanced user to customize one's sentinel. This section allows the user to customize the following features.

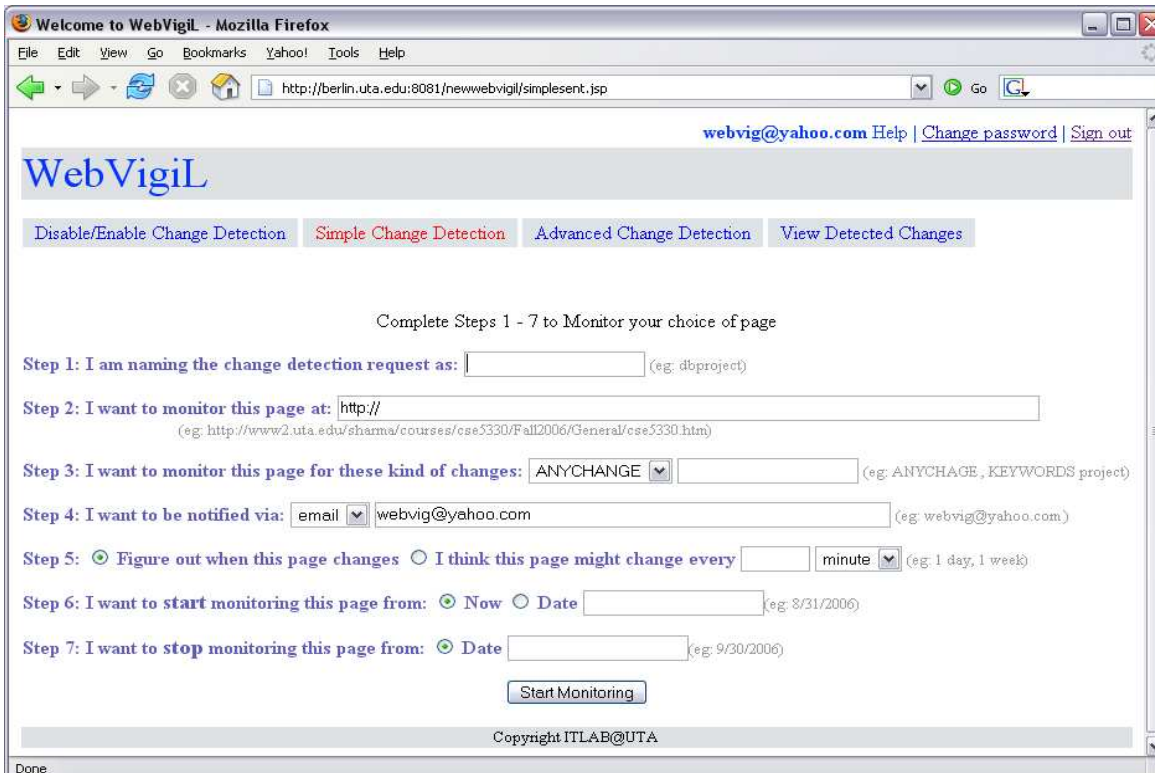

Figure 6.2 Simple Change Detection

- The user can use the change types and operators to build an infix expression for placing sentinel with a composite change type. For example (ANYCHANGE AND NOT IMAGES) AND (NOT LINKS OR KEYWORDS [bomb, threat]) is an infix expression that the user can build using the menus.
- The frequency with which the user intends to receive notifications. The available options are 'BEST EFFORT', 'IMMEDIATE', 'INTERACTIVE' and 'TIME INTERVAL'. BEST EFFORT AND IMMEDIATE allows the notification to be sent whenever the page changes. But INTERACTIVE option allows the user to come to user interface and manage their sentinel. whereas the TIME INTERVAL option allows the users to customize the number of notifications received.

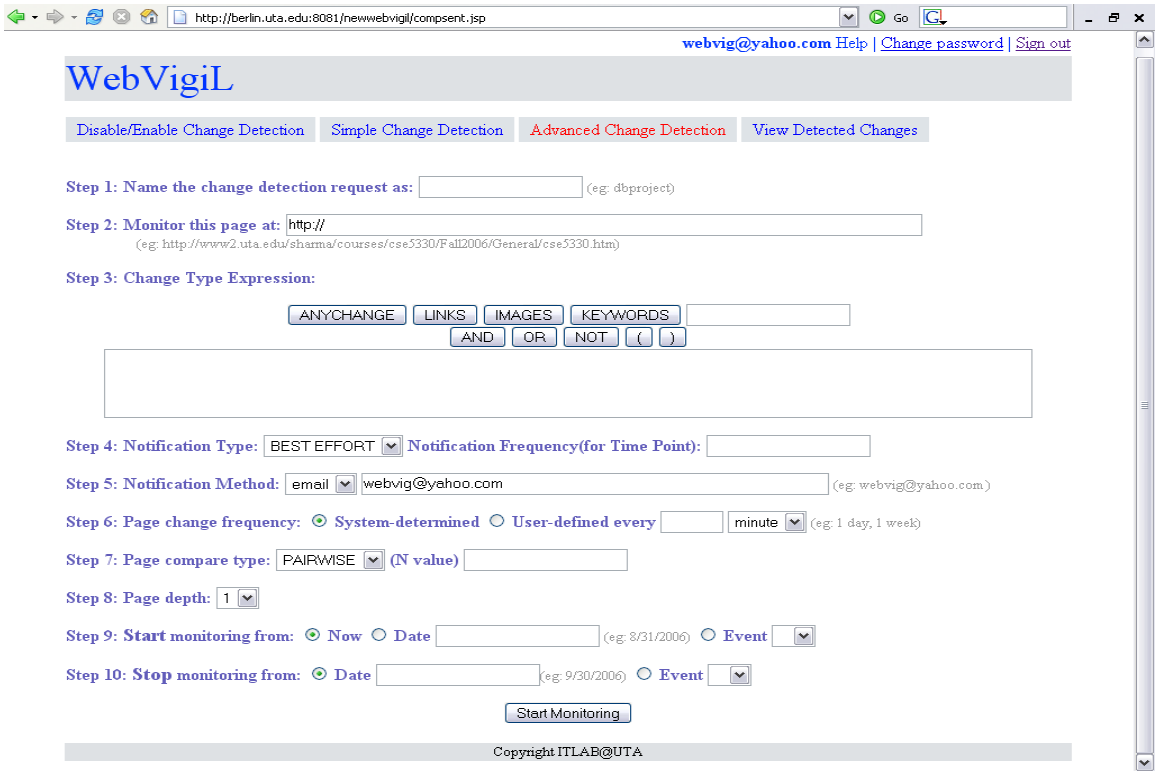

Figure 6.3 Advanced Change Detection

- The users are allowed to choose the compare options, WebVigiL uses to detect changes. The available options are 'PAIRWISE' and 'MOVING N'.
- The users can also specify the depth of the website where change has to be detected.
- Finally the user also has the flexibility to start and end their monitoring request at current time or at a specific date (time point) or with respect to start or end of their previously defined sentinels.

These customization allows the user to set complex sentinels and receive notification. This is an example of selective and customized change detection supported by WebVigiL. Figure 6.3 shows a snapshot of Advanced change detection request interface.

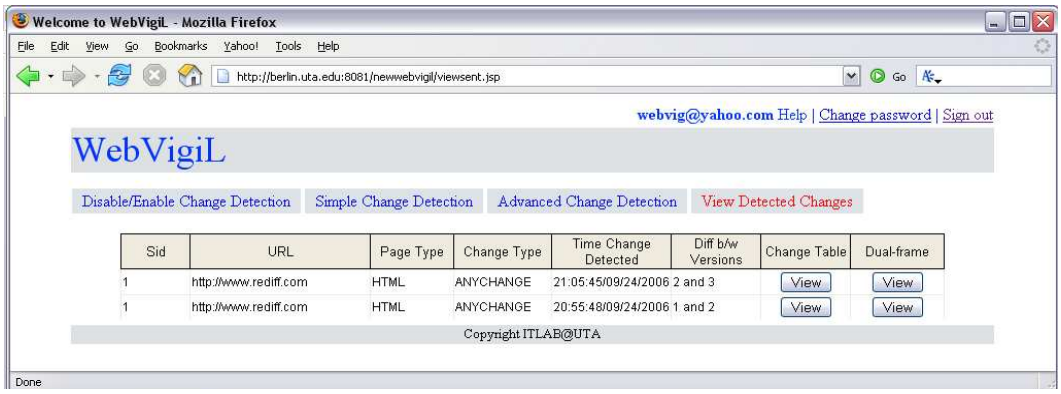

Figure 6.4 View Changes

6.1.4 View Detected Changes:

DashBoard allows the users to view detected changes from their sentinels. This section was specifically designed so that the users may not want to be notified by mails and is willing to log into WebVigiL interface to view the detected changes. The Interface lists all the sentinels with their detected changes in the form of a change table or dual-frame format. The user can choose the type of view and click the corresponding button to view it. The user can also sort the listing according to the change detection time and view the most recent change detected. The list displays only the last 3 detect changes, which reduces the amount of effort, the users might need put up to sort through the mails or browse through all the changes that were detected. Figure 6.4 shows snapshot of the interface.

6.2 Improvements in change presentation

Current change detection is restricted to HTML and XML documents in the system. XML does not contain presentation tags whereas in HTML the data is embedded in the presentation tags. The tags in XML define the content. Hence the change presentation of HTML and XML is handled in a different manner. The rest of the section contains a

detailed discussion on the presentation schemes used in WebVigiL for change presentation in HTML and XML separately.

6.3 Content in Notification

The presentation email contains the following basic information for users to be able to clearly perceive the computed differences in the context of one's predefined specification.

- URL for which the change detection was invoked
- The type of change (for e.g., Keywords, Phrases)
- Notification time
- A Link for viewing Changes Table
- A Link for viewing Dual Frame Presentation

The content of the notification has been designed to be as small as possible to satisfy the network quality of service requirements.

6.4 Change Presentation for HTML Pages

In WebVigiL, different types of customized changes such as keywords, phrases etc are supported, which need to be represented and displayed in different ways. Currently the types of changes supported by the system are ANYCHANGE, LINKS, IMAGES, KEYWORDS and PHRASES. This section elaborates on the two schemes used for presenting changes in HTML pages.

6.4.1 Change-Only Approach

The changes are represented in a tabular structure with the type of change whether it is an insert/delete/update. If the user merely wants to know of the changes and not the context with respect to the changes, this approach is very useful. If the sentinel is

of change type 'links OR images', the table shows the source of the link or image and whether it is an insert or a delete. For keywords, the table contains the keywords if it is an insert or a delete. Presentation of this type is advantageous when the number of changes are large and there is a lot of content in the page. The amount of data sent in this approach is minimal which is appropriate for devices with less storage and when the bandwidth is small.

6.4.2 Dual-Frame Approach

This Approach shows both the documents side-by-side in different frames highlighting the changes between the documents. The benefit of this type of presentation is that it is visually pleasing to the user for easy interpretation as the contents of the page along with the changes highlighted are displayed. The various issues involved in highlighting and displaying the HTML pages are as follows.

- Location information is not provided for HTML pages
- Along with content there is script and tags
- Difficult to localize keywords if there is more than one occurrence of the keyword

The first problem can be solved by using a regular expression. Since reading line by line to find the keywords and highlighting them would be rather difficult and time consuming if there is a lot of content to parse. Since there is no location information, every keyword in the page will be highlighted. The tags and scripts are replaced with special characters before highlighting the content. Once highlighting is done, the scripts and tags are put back to bring back the original page. Figure 6.5 shows Dual-frame representation for HTML pages.

Figure 6.5 Dual-Frame representation of ANYCHANGES

6.5 Change Presentation for XML Pages

A HTML browser is able to render the HTML document because the browser has prior knowledge of how to render a specific HTML tag. The browser applies style rules to predefined 'tags'. For instance, the contents of the $< H1 >$ element might be displayed as a block with line breaks preceding and following the text which is put in $18pt$ bold Times New Roman. This is not possible in XML as the user defines the tags. Thus, the browser may not know how to render user defined tags. There are three ways to instruct the browser on displaying the tags.

- XML + CSS
- XML + XSL (Extensible Style Language)
- Convert XML to HTML

```
<?xml version="1.0"?>
<?xml-stylesheet href="test.css" type="text/css" ?>
<addrbook>
 <entry>
   <name>Dr. Shrama Chakravarthy</name>
   <address>
     <address1>CSE DEPARTMENT</address1>
     <address2>Univ. of Texas</address2>
     <address3></address3>
     <city>Arlington</city>
     <state>TX</state>
     <zipcode>76019</zipcode>
   </address>
 </entry>
 <entry>
     <name>Dr. Raman Adaikalavan</name>
     <address>
       <address1>CSE DEPARTMENT</address1>
       <address2>Univ. of Texasl</address2>
       <address3></address3>
       <city>Arlington</city>
       <state>TX</state>
       <zipcode>76019</zipcode>
     </address>
      </entry>
</addrbook>
```

Figure 6.6 XML page

```
entry
{
display: block; padding-left: 5px;
 padding-right: 5px; padding-top: 5px;
 padding-bottom: 5px; color: white;
 background: black; font-family: times new roman;
 font-size: 12pt; font-weight: bold;
}

name
{
 display: block; color: white; background: red;
 font-family: arial; font-size: 18pt; font-weight: bold;
 text-align: center;
}

address1, address2, address3,
homePhone, workPhone, fax, email,
url, comment
{display: block;}

city, state, zipcode
{ display: inline;}

state
{ text-transform: uppercase;}
```

Figure 6.7 CSS for XML page

A major design goal of XML was to separate the document's content from its presentation. The XML document can be seen as the container for the information content with an external style sheet functioning as the processing instructions for presentation. The two main style sheets languages for XML are Cascading Style Sheets (CSS) and Extensible Style Language (XSL).

A XML document fetched from the web based on users sentinel, may or may not have a style-sheet linked to it or the style-sheet may not even be publicly accessible, making it difficult to be fetched along with XML document. In such cases, a new custom made style sheet has to be created or the existing style sheet has to be modified to highlight the detected changes. This presents problems of its own. As an example, Figure 6.6 shows an address book entry for doctors in University of Texas and Figure 6.7

shows the CSS associated with the page. If one of the doctors changes university or the address changes then there is no way to specify that in the style sheet. Since the style sheet shows all the details of the address book entry in same style, selective highlighting for presenting detected changes becomes difficult. Similar problem is faced when XSL is used for presenting the XML page. Thus the third approach of converting the XML to HTML page is used for presenting the changes in WebVigiL. At present, two schemes are used to present the changes in an elegant and easy to understand manner. The schemes along with their advantages and disadvantages are discussed below:

6.5.1 Changes-Only Approach

In this approach only the changes are presented. The traditional method is a tabular structure with the types of changes (insert/delete/move) as different columns of the table. The changes presented may be difficult for the user to decipher, as these changes are not to be shown relative to the original document. The tabular presentation shows the content of the change, count of how many time the keywords of interest appear in the old and the new page along with the signature of the node involving the change.

6.5.2 Dual-Frame Approach

In this approach we show both the documents side-by-side to highlight the changes. This approach has an advantage over all the other approaches of being very easy to interpret. But in this case, we need to embed them in HTML to highlight the changed contents. WebVigiL uses the above approach because of the following issues.

- No unique style sheet for all kinds of XML documents
- The style sheet embedded in the XML document is not always downloaded along with the XML page

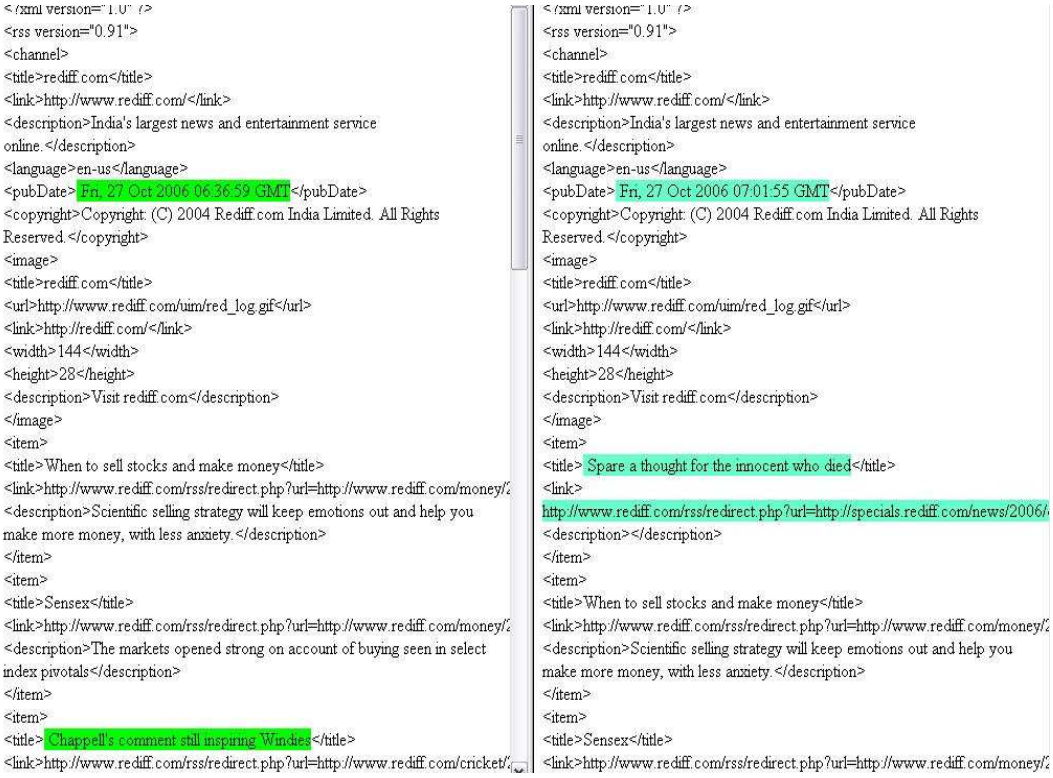

Figure 6.8 Dual-Frame representation of ANYCHANGES

Embedding the XML in HTML page is done by Adding '~!~' at end of every line then replace '<' with < , '>' with > and '~!~' with a $< br >$. Doing this converts the XML page to a content which can used by the regular expression to highlight the changes. Now the usual method followed for scanning the keywords and highlighting in HTML is used. Now the content is placed inside a HTML structure to make it complete HMTL page which can present changes in XML. Figure 6.8 shows Dual-frame representation for XML pages.

6.6 Improving Change Detection

CH-DIFF [3] which does customized change detection of HTML pages requires a HTML parser to achieve its objective. Quiotix HTML parser was used to achieve the

task of extracting the object from the HTML page for change detection. Since the parser was not able to handle dynamic pages , embedded tables and tags like $< DIV >$ and $< SPAN >$ all the relevant information from the HTML page could not be parsed. Thus change detection on two different pages on same URL did not produce the required result. This section discusses the shortcomings of the old parser and the advantages of the new parser.

6.6.1 Quiotix HTML Parser:

Quiotix HTML Parser [30] is a JavaCC grammar for parsing HTML documents. It builds a simple parse tree, which is used to validate, reformat, display, analyze, or edit the HTML document. The parser produces a parse tree of the information contained in the source file, and hence dumping of the parse tree results in an almost identical copy of the input document. The generated parse tree supports the commonly used "Visitor" design pattern. Several visitor classes have been provided, which do things like dump the parse tree, restructure the parse tree, etc. Common tasks such as formatting, validation, or analysis are easily performed as Visitors. The drawbacks of this parser are as follows: This parser was using a lot of memory and was causing the program to run out of memory. It truncates the contents, if it is not able to understand the HTML page. It also was not designed to handle nested table, $< div >$ and $< span >$ tags.

6.6.2 HTMLParser:

HTML Parser [31] is a Java library used to parse HTML in either a linear or nested fashion. Primarily used for transformation or extraction, it features filters, visitors, custom tags and easy to use JavaBeans. It is a fast, robust and well tested package. The new parser was able to handle almost all of the dynamically generated pages. The

parser was able to parse through contents which were nested deep inside tables and also handled the new tags efficiently.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

WebVigiL is a change monitoring system for the web that supports specification, management of sentinels, change detection for HTML and XML pages, and provides presentation of detected changes in multiple ways. It is a complete system that allows monitoring and notification of changes to structured documents in a distributed environment. The contributions of this thesis are the following:

- Design and Development of a Load balancer for WebVigiL
- A web-based user interface (DashBoard) for the user to manage their sentinels
- Presentation techniques to display detected changes for HTML and XML pages in a dual-frame format
- Improving the change detection accuracy
- WebVigiL system integration and testing for various test cases

With the addition of a load balancer, WebVigiL will be able to use multiple servers and handle a large number of sentinels. The users will also be able to manage their sentinels from a single GUI. They will also be able to view changes to their sentinel in A more pleasing and easy to understand format. WebVigiL is also more accurate in detecting changes because it is now able to parse more nuances of HTML pages.

7.2 Future work

WebVigiL's scalability has been improved, but still there is scope for improving the performance and robustness of load balancing by including a failure recovery for load

balancer. The amount of resources used can also be reduced by decreasing the number of databases and web servers used. Failure recovery can also be made transparent to the users by transferring the sentinels to a live server when a server goes down. The load balancer can also be made intelligent by learning URL's pattern in sentinels, so that a server can be dedicated for those kind of URLs (Example: sports URLs in one server, news URLs in one server).

REFERENCES

[1] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive push-pull: Disseminating dynamic web data," in *Proceedings of the Tenth International WWW Conference*, Hong Kong, China, 2001.

[2] S. Chakravarthy *et al.*, "Webvigil: An approach to just-in-time information propagation in large network-centric environments," in *Second International Workshop on Web Dynamics*, Hawaii, 2002.

[3] N. Pandrangi, "Webvigil: Adaptive fetching and user-profile based change detection of html pages," Master's thesis, The University of Texas at Arilngton, 2003.

[4] A. Sanka, "A dataflow approach to efficient change detection of html/xml documents in webvigil," Master's thesis, The University of Texas at Arilngton, 2003.

[5] J. Jacob, "Webvigil: Sentinel specificatin and user-intent based change detection for xml," Master's thesis, The University of Texas at Arilngton, 2003.

[6] A. Sachde, "Persistence, notification and presentation of changes in webvigil," Master's thesis, The University of Texas at Arilngton, 2004.

[7] S. Chamakura, "Improvements to change detection and fetching to handle multiple urls in webvigil," Master's thesis, The University of Texas at Arilngton, 2004.

[8] A. Eppili, "Approaches to improve the performance of storage and processing-subsystems in webvigil," Master's thesis, The University of Texas at Arilngton, 2004.

[9] L. Ling, P. Calton, and T. Wei, "Webcq: Detecting and delivering information changes on the web," in *Proceedings of International Conference on Information and Knowledge Management (CIKM)*, Washington D.C, 2000.

[10] Watch-That-Page, "http://www.watchthatpage.com/."

[11] WisdomChange, "http://www.wisdomchange.com/."

[12] ChangeDetection, "http://changedetection.com/."

[13] ChangeDetect, "http://changedetect.com/."

[14] TrackEngine, "http://www.trackengine.com/."

[15] Copernic-Tracker, "http://www.copernic.com/en/products/tracker/."

[16] Website-Watcher, "http://aignes.com/."

[17] Server-Load-Balancing, "http://content.websitegear.com/."

[18] T. Bourke, *Server load balancing.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.

[19] Cisco's-Local-Director, "http://www.cisco.com/."

[20] BIG-IP, "http://www.f5.com/products/bigip/."

[21] N. Pandrangi, J. Jacob, A. Sanka, and S. Chakravarthy, "WebvigiL: User profile-based change detection for HTML/XML documents," in *New Horizons in Information Management, 20th British National Conference on Databases, BNCOD 20, Coventry, UK, July 15-17, 2003, Proceedings*, vol. 2712. Springer, 2003, pp. 38–57.

[22] A. Eppili, J. Jacob, A. Sachde, and S. Chakravarthy, "Expressive profile specification and its semantics for a web monitoring system," in *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 2004, Proceedings.* Springer, 2004, pp. 420–433.

[23] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite events for active databases: Semantics, contexts, and detection," in *Proceedings, International Conference on Very Large Data Bases*, 1994, pp. 606–617.

[24] R. Dasari, "Design and implementation of a local event detector in java," Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, 1994.

[25] S. Yang, "Formal semantics of composite events for distributed environments: Algorithms and implementation," Master's thesis, The University of Florida, Gainsville, December 1998.

[26] H. Lee, "Support for temporal events in sentinel: Design implementation and preprocessing," Master's thesis, The University of Florida, May 1996.

[27] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data and Knowledge Engineering*, vol. 14, no. 10, pp. 1–26, October 1994.

[28] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra, "Design of sentinel: An object-oriented dbms with event-based rules," *Information and Software Technology*, vol. 36, no. 9, pp. 559–568, 1994.

[29] S. Chakravarthy *et al.*, "A learning-based approach for fetching pages in webvigil," in *Symposium On Applied Computing*, March 2004.

[30] Quiotix-Parser, "http://www.quiotix.com/downloads/html-parser/."

[31] HTMLparser, "http://htmlparser.sourceforge.net/."

BIOGRAPHICAL STATEMENT

Chelladurai Hari Hara Subramanian was born in Tuticorin, TamilNadu,India in 1981. He received his B.E. degree from Bannari Amman Institute of Technology, Coimbatore, India in 2003 and his M.S. degree from The University of Texas at Arlington in 2006, both in Computer Science and Engineering. During his period of study he worked as an intern in General Electric, Atlanta, Georgia. He also worked as System Administrator for the CSE Department, The University of Texas at Arlington. His research interests include active databases and web technologies.