PERSISTENCE, NOTIFICATION AND PRESENTATION OF CHANGES IN WEBVIGIL

The members of the Committee approve the master's thesis of Alpa Sachde

Sharma Chakravarthy
Supervising Professor _____

Alp Aslandagon _____

Lynn Peterson _____

To my Family and Friends.

PERSISTENCE, NOTIFICATION AND PRESENTATION OF CHANGES IN WEBVIGIL

by

Alpa Sachde

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

May 2004

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for his constant guidance and support, and for giving me a wonderful opportunity to work on such a challenging project. I am also grateful to Dr. Alp Aslandogan and Dr. Lynn Peterson for serving on my committee.

I would like to thank Ajay Eppili and Shravan Chamakura for helping me throughout the design and implementation of this work. Thanks are due to my seniors Jyoti Jacob and Naveen Pandrangi and Anoop Sanka for their fruitful discussions and guidance. I would like to thank my roommate Swapna Rao and my friends Sridhar Varakala, Hima Valli Kona and others for their help. I also thank my friends L.Stephen Lobo, Laali Elkhalifa and Raman Adaikkalavan and Manu Aery for their support and encouragement in ITLAB.

I would like to acknowledge the support, by the Office of Naval Research & the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-2-05430), and by NSF (grant IIS-0123730) for this research work.

I would like to thank my friend, Himanshu Doshi for their guidance, support and patience.

I would also like to thank my parents and brother for their constant love and support throughout my academic career.

April 8, 2004

ABSTRACT

PERSISTENCE, NOTIFICATION AND PRESENTATION
OF CHANGES IN WEBVIGIL

Publication No. _____

Alpa Sachde, M.S.

The University of Texas at Arlington, 2004

Supervising Professor: Sharma Chakravarthy

The World Wide Web is an omnipresent and ever-expanding source of data. The exponential increase of information on the web has affected the manner in which it is accessed, disseminated and delivered. The emphasis has shifted from mere surfing the web for information to efficient retrieval and monitoring of selective changes to its content. Hence, an effective monitoring system for change detection and notification based on user-profiles is needed. WebVigiL is a general-purpose, active capability-based information monitoring and notification system. It handles specification, management, and propagation of customized changes as requested by a user.

As change detection requires two versions of a page for comparison, and the same page is retrieved multiple times over a period of time, there is a need for storing, retrieving, and managing persistent versions of web pages of interest. This thesis presents an efficient mechanism to store, manage, and provide the appropriate pages as needed to perform change detection. It also deals with purging the versions when they are no longer needed. This thesis also deals with timely notifications of customized changes to users.

It addresses various alternatives for notifying the users when change is detected. Finally, this thesis addresses presentation of changes as it is difficult to manually locate changes in a page. The changes are highlighted in order to present them in an easy-to-understand manner.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 World Wide Web

The Internet is evolving as an indispensable repository of shared information. The explosive growth of information on the web has raised a need for fresh ideas on information search and its delivery. In this context the emphasis has shifted from mere viewing of information to efficient retrieval and monitoring of selective changes to information content in a timely manner. The sheer size of information on the web has affected the manner in which information is accessed, disseminated and delivered. The rapid growth of static pages has been projected to be 15% per month. Nowadays different users are interested in different levels of granularity of a specific page instead of the whole page. For example, some are interested in the appearance/disappearence of some keywords or phrases and some others may be interested in any change to a page. The need to monitor changes to documents of interest is not only true for the internet but also for other large repositories. In general, the ability to specify different types of changes to arbitrary documents and get notified according to user-preferred ways in a timely manner is required to reduce or avoid the wasteful navigation on the web and its associated cost in the information age.

1.2 Existing Paradigms

Today's web usage follows two approaches, namely, the pull paradigm and the push paradigm. Change detection using the former approach entails polling or posing appropriate query to the web server (where the page of interest resides) for selective

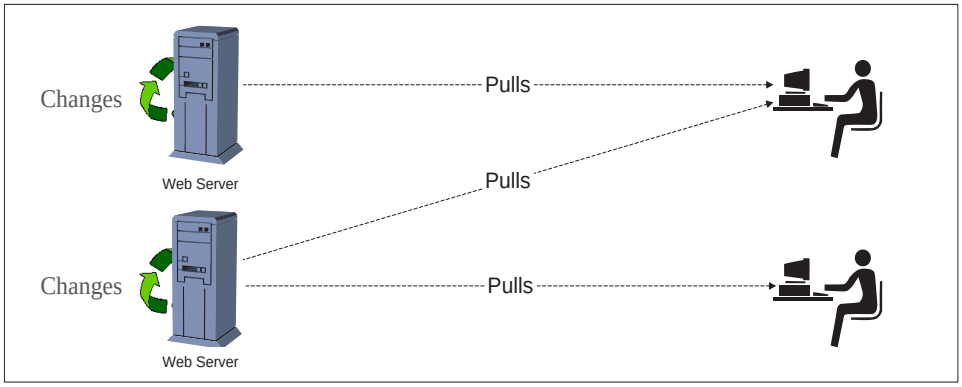


Figure 1.1 Pull Paradigm.

changes. In the later approach the web server pushes the information of interest to the subscribed users.

1.2.1 Pull Paradigm

As the information needed by the user is distributed across large repositories, it is a cumbersome task for the users to pull the pages of interest periodically [1]. As we can see in Figure 1.1 users have to seek the information of interest. If they are interested in specific changes, the burden is on them to find for the same manually from the corpus of information on the pages. The frequency with which the user need to retrieve the pages to determine that a change has occurred has to be determined by the user. Based on the frequency, it is likely that the users may miss some changes. The above process results in wasteful navigation of web. It is very labor intensive due to the human in the loop.

1.2.2 Push Paradigm

On the other hand, in push technology [1], the users do not have to query the system periodically, but they specify their interests. To follow this paradigm, the system has to accept user intent and should have a mechanism detect user intent and inform the user in a timely manner. As shown in Figure 1.2 the system notifies the users with

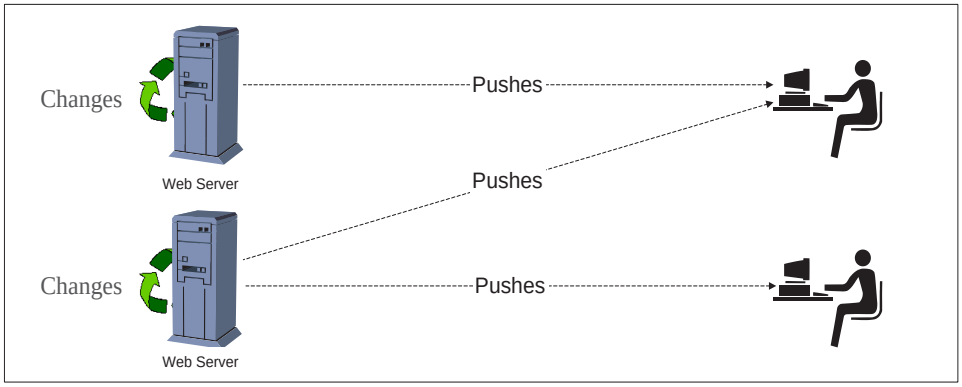


Figure 1.2 Push Paradigm.

the relevant information in a timely manner. A typical scenario is that of a mailing list which is used to send information periodically to all the subscribed users. For instance, American Airlines [2] uses a mailing list to send information to send the NetSaver Fare Alerts to all the subscribers on list. Typically, the same information is sent to all subscribers. Though interested in a particular source or destination, users might receive unnecessary information of all the fare sales which is not desired. The above results in wasted bandwidth and does not serve the purpose of the users of finding the relevant information. The emphasis in this thesis is on selective change detection, as users are typically interested in changes to a particular portion or section of a page and may not necessarily for the entire page.

Other tools that provide real-time updates in the web context (e.g., stock updates) are customized for a specific purpose and have to be running continuously. And underneath these system still use a naïve pull paradigm to refresh the screen periodically.

1.3 Motivation

From the above it is clear that there is a need for a generalized system that checks the pages of interest for changes and notifies the users based on certain specified profiles.

The users can subscribe to a page and specify changes of interest. They can receive customized information of the changes in content they are interested in. The quality of service requirements such as timely notifications need to be considered as well. Most of the systems developed so far that support selective notification are highly domain specific. The proposed system, WebVigiL is an efficient and effective general purpose change-detection and notification system [3]. It uses an appropriate combination of the push and pull paradigms as shown in Figure 1.3. It uses an intelligent pull paradigm to retrieve information from remote sources and uses a push technology to notify users of changes of their interest in structured and semi-structured documents. It is a general-purpose, active capability-based information monitoring and notification system. It handles specification, management of sentinels and propagation of information requested by sentinels. Sentinels are user profiles/policies provided by the users to specify their interests. The focus of change detection in WebVigiL is to detect selective changes on the content of the document, based on user intent. Currently HTML and XML documents are supported. The framework and architecture does not change to use it for other types of documents. It has been modularized in such a way that only the change detection modules need to be replaced if other types of documents need to be supported.

Active Capability is a well-established paradigm in the context of databases. There are many database applications that require timely notification, for which active rules have been proposed as a paradigm to support their requirements. For the active capability to be used on a broader scale, for applications such as WebVigiL, the scope needs to be extended beyond the context of databases. (Event-Condition-Action Rules (or ECA) rules are used to capture the active capability of the proposed system.

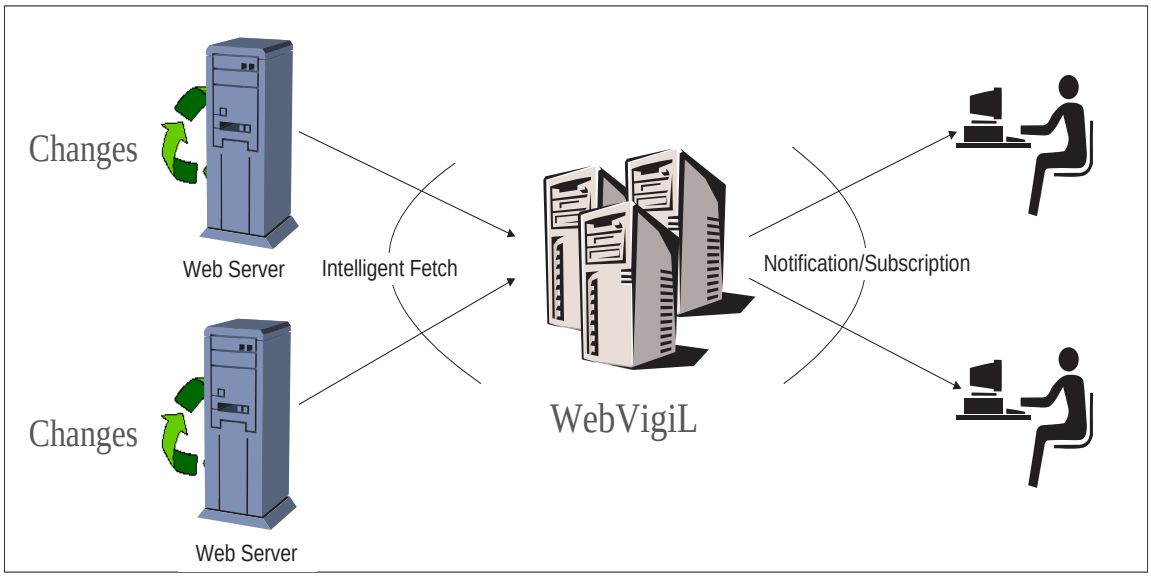


Figure 1.3 WebVigiL Motivation.

1.4 Sample Applications

Some of the sample applications where a system such as WebVigiL will be useful are listed below:

Monitor Software Development Documents: In large software development projects, there exist a number of documents, such as requirements analysis, design specification, detailed design document, and implementation documents. The life cycle of such projects is in years (and some in decades) and changes to various documents of the project take place throughout the life cycle. Typically, a large number of people are working on the project and managers need to be aware of the changes to any one of the documents to make sure the changes are propagated properly to other relevant documents and appropriate actions are taken. Large software developments happen in distributed environments.

Monitor Discussion Boards: Nowadays there are discussion groups for almost every topic that we can think of. The discussion boards of such groups can be monitored

for the update of any new posts on the topic of interest. Students may want to monitor the class discussion boards. Students may want to know when the web contents of the courses (they have registered for) change.

News Updates: Users may want to know when news items are posted in a specific context(appearance of key words, phrases etc.) they are interested in.

Monitor for Security: A website administrator can monitor his/her web pages for any unintended changes from an intrusion/security point of view.

Information Filtering: In addition to all of the above, the system can be extended to include information filtering wherein the changes will be detected on pages when a specified pattern of text appears.

WebVigiL architecture is modular and extensible. So it can accommodate the needs of any application with very few changes such as in the change detection module. The change detection of the document may vary a little bit and almost all the other modules remain unchanged.

1.5 Contributions

Various contributions of this thesis are described below. A mechanism to maintain a central repository of pages that archives, manages and provides the versions of pages needed by various modules in the WebVigiL system has been designed and implemented. The design and implementation of a notification module that delivers timely notifications to the users is presented in this thesis. Various schemes for presenting changes in HTML and XML documents in way that facilitates the users to interpret the changes in an intuitive manner are discussed. In addition this thesis also discusses the design and implementation of an interactive user interface that accepts user input, validates it and sends it to the system to monitor the pages of interest to the users.

The thesis is organized as follows, chapter 2 gives an overview of the architecture of WebVigiL and its various modules and chapter 3 discusses a mechanism to notify changes, chapter 4 addresses the issues of how the versions involved in the change detection are archived and managed, chapter 5 outlines different ways of presenting the change to HTML and XML documents, chapter 6 gives the details of implementation of all the modules that are contributed in this thesis, chapter 7 describes in short about the related work and finally chapter 8 outlines conclusions and future work.

CHAPTER 2

WebVigiL Architecture

WebVigiL is a general purpose change detection and notification system that monitors changes to unstructured documents. Currently the work addresses HTML and XML documents. The objective of the system is to facilitate selective change detection. Various modules in the system help to specify, manage and propagate user requests to monitor web pages at different levels of granularity [4]. The system comprises of various modules, which constitute the architecture. The rest of the chapter briefly describes various modules of WebVigiL [3]. Figure 2.1 summarizes the high level WebVigiL Architecture.

2.1 Sentinel

Users specify their monitoring request in the form of a sentinel. A web based user interface is provided to the users to submit their profiles termed as sentinels. The users are required to give certain details for placing a monitoring request with WebVigiL system which are as follows:

- Page to be monitored (URL).
- Type of content to be monitored.
- Frequency by which the system checks for changes (either user specified or left to the system)
- Start and End time for the monitoring request.
- The medium and frequency for notification of changes.
- The relative version of comparison for changes.

WebVigiL provides an expressive language with well-defined semantics for specifying the monitoring requirements [5]. The specification language supports a number of features some of which are summarized below:

As only monitoring any change to a page as a whole may be too restrictive, we support a set of change types for monitoring a page at different levels of granularity in WebVigiL. The set includes changes to images, changes to links, changes to specified keywords or phrases. The users can even place a request with the above mentioned change types or a combination of them (for example changes to links and images). The system also allows the users to set sentinels on their own previously defined sentinels which provides a way to keep track of correlated changes. Unless the users explicitly specify the properties, the new sentinel inherits all the properties of the previous sentinel. The system provides a way of specifying the fetch frequency which is the frequency by which the users want the system to check for changes in the page of interest. The options are either a user defined frequency (e.g., every 2 days) or system defined frequency where the system tries to tune the fetch frequency to the actual change frequency using a learning algorithm based on history. The medium of notifications that can be specified are email, fax, PDA and dashboard. Currently email and dashboard are supported. The frequency of notification can be given as user-specified frequency 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 every 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 the last two version of the page are compared.

For example consider the Scenario: Smith wants to monitor changes to links and images to the page "http://www.msn.com" and desires to be notified daily by e-mail starting from April 8, 2004 to May 8, 2004. The sentinel specified for the above scenario is as follows:

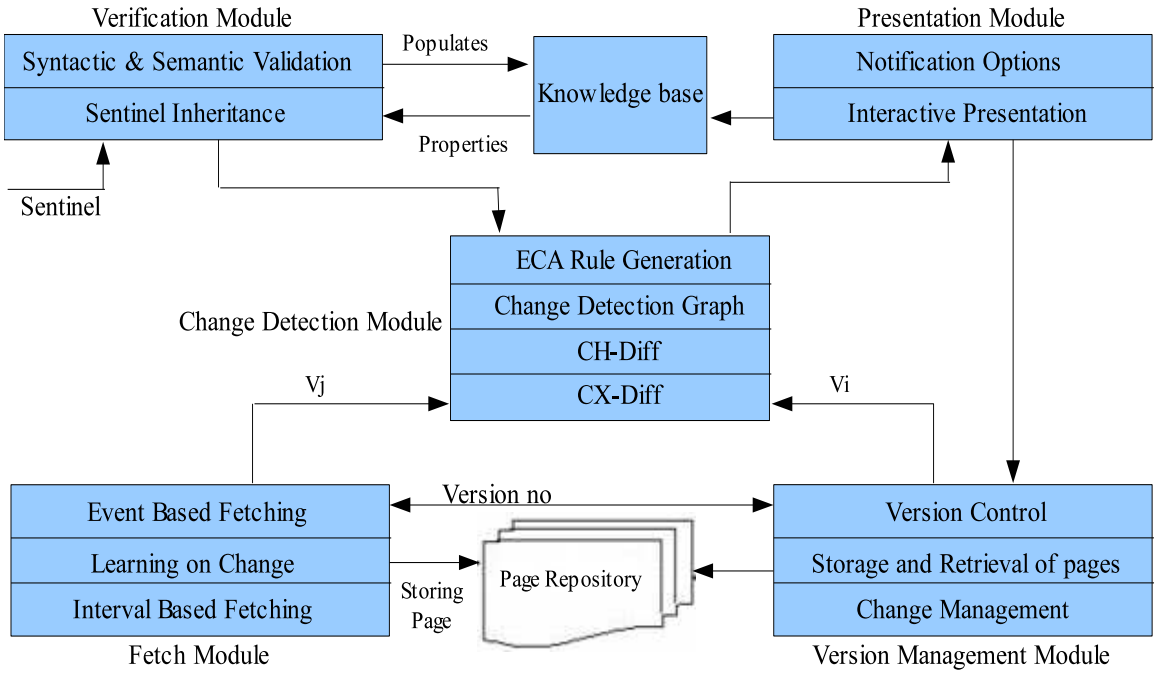


Figure 2.1 WebVigiL Architecture.

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

Monitor all links AND all images

Fetch every 2 days

From 04/08/04 To 05/08/04

Notify By email smith@msn.com Every 4 days

Compare pairwise

2.2 Validation

As the sentinels installed on WebVigiL system are required to follow the strict specifications, given input is validated. Once the users provide the required input through the web interface, the input is first validated on the client side. Most of the validations are completed on the client side. This reduces the communication with the server for validations. Only those validations that depend on the information stored on the server

are done on the server side, once the sentinel is sent to the server side. For example the start and end times of inherited sentinels are checked. For example 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 an error is thrown to the user. The sentinel name is checked for duplication for the same user so that every sentinel installed by the same user has a unique name.

2.3 KnowledgeBase

Once the user input(sentinel) is validated, the details of the sentinel are stored and persisted in a repository which is known as KnowledgeBase. Knowledgebase are relational tables in a database. The database used currently is Oracle. The details of the users are also stored in this repository. As there are several modules that require the information of a sentinel at run time, it was essential for the metadata of the sentinel to be stored in a persistent recoverable manner. 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 instance the change detection module detects changes based on sentinel information such as the URL to be monitored, the change and compare specifications, and the start and end of a sentinel. The fetch module fetches the pages based on the user specified fetch policy. The notification module requires appropriate contact information and notification mechanism to notify the changes. User information, such as the sentinel installation date, and the page versions for change detection and storage path of detected changes also need to be stored to allow a user to keep track of his/her sentinels. The database is updated with important run time parameters by several modules of the system. The change detection module updates the respective relations with the status of the sentinel and recently detected changes for notifying the

users. Queries are posed to retrieve the required data from the tables at run time with the help of a JDBC bridge.

2.4 Change Detection Module

Nowadays as the web is burgeoning at an astounding rate, there is a paradigm shift on how web is used. Users now look for selective changes on the web. WebVigiL system provides that through the change detection module.

2.4.1 ECA Rule Generator

In WebVigiL, each valid request that arrives induces a series of operations that occur at different points of time [6, 7]. 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 rule generation module generates Event Condition Action (ECA) rules [8, 9] 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 [10, 11] along with the local event detector (LED) [6, 7] are used:

1. To generate fetch rules for retrieving pages.
2. To detect events of interest and propagate pages to detect primitive and composite changes.
3. To perform activation and deactivation of sentinels.
4. For time-based notification of changes.

Activation/Deactivation: WebVigiL accepts interval-based monitoring requests. Hence, each sentinel has a start and end time during which a sentinel is enabled by de-

fault. 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 [8] are defined and rules are associated for fetching with the periodicity as 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 in which the users specify the frequency desired for notification of changes. Again, we use ECA rules to perform the above. The system registers with the LED and creates periodic events [8] 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 detected since the last notification and notifies the corresponding user (action part of the rule) if a change has occurred. A notify rule is created and used to send time-based notifications to users.

2.4.2 Change Detection Graph:

When there are two or more sentinels interested on a particular change type on the `same` page, preferably, the change should be computed only once. The above problem is overcome by grouping the sentinels interested in the same change type on the same

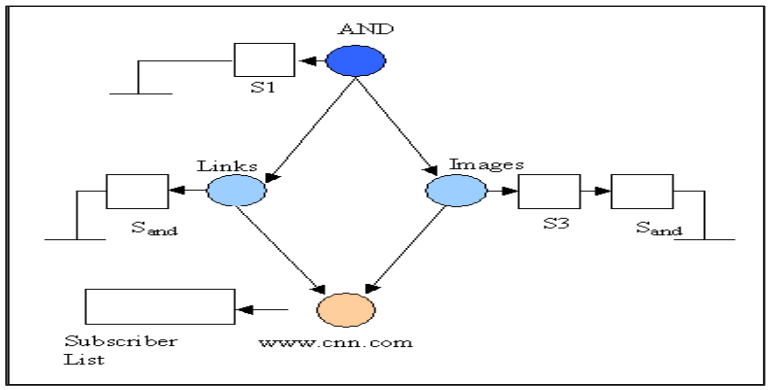


Figure 2.2 Change Detection Graph.

page. The relationship between the sentinels and page is captured using a graph [12]. The graph aids the change detection module in being less computationally expensive 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 2.2.

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 references by sentinels in the system at any point of time.

Change type node: All the nodes one level above the leaf node (i.e level-1 nodes) in the graph are change type nodes (images, links). These nodes represent the type of change specified on a page. The change types nodes supported by the system are: all words, links, images, keywords, phrases, table, list, regular expression, and any change. item[Composite Node:] A Composite node represents a combination of change types. For example "all links and all images". All higher-level nodes (i.e., greater than level-1) in the graph belong to this type. Currently, we support

composite changes on a single page. We are investigating into extending the system to support multiple pages for the same sentinel.

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 of each node. The list corresponding to URL node consists of all the change type nodes that have 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 happens to have changed the sentinels interested in 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 in 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.

2.4.3 Change Detection (CH-Diff & CX-Diff):

Currently, change detection has been addressed for Hypertext Markup Language (HTML) and eXtensible Markup Language (XML) documents which are the two most popular forms for publishing data on the web. Change detection algorithms [4, 5] work as per the change type specified by the users while giving the input. The corresponding objects are extracted from the versions being compared. The versions to be compared

for change detection are decided by the option 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.

2.5 Fetch Module

The fetch module fetches the pages of interest to the users as registered with the system while setting up a sentinel. The fetch module retrieves the page based on the specified frequency. The options for the frequency are user-defined frequency (for e.g., every 2 days) or can be system-defined. This module thus serves as a local wrapper for the task of fetching pages depending upon the user-set fetching policy. This module informs the version controller of every version it fetches, stores it in the page repository and notifies the CDG of a successful fetch. The properties of a page are checked for freshness in a page. The properties that a page has are: Last Modified Date (LMT) or size of the page (Checksum). Depending upon the nature (static/dynamic) of page being monitored the complete set or subset of the meta-data is used to evaluate the change. Only if the properties have changed, the wrapper fetches the page and sends it 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 fetch frequency to the actual change frequency of a page. BE Rule uses a best-effort Algorithm (BEA) [13] to achieve this tuning. In the best-effort algorithm, the next fetch interval (Pnext) is determined from the history of changes to that page. When the next polling interval is determined, the

BE Rule changes the interval "t" of the periodic event.

Interval-Based (IB) Rule: The user explicitly provides a fetch frequency. A periodic event [8, 9] with periodicity (interval t) equal to the given interval is created and an IB rule is associated with it to fetch the page. As a result there will be more than one IB 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).

2.6 Version Management Module

As WebVigiL performs the change detection by comparing two versions of a page of interest, there is a need for a repository service (Version Manager) where the versions are archived and managed efficiently. It archives, manages and supplies different modules with the required versions at run time. The primary purpose of the repository service is to reduce the number of network connections to the remote web server by avoiding unnecessary fetches, thereby reducing network traffic. The quality of service requirements for the repository service include managing multiple versions of pages without excessive storage overhead. So there is a necessity to store only those versions needed by the system and purge all the versions not useful. The details of this module will be discussed in chapter 4.

2.7 Presentation & Notification Module

One of the important modules of web monitoring is presentation and notification module. The primary functionality of this module is to notify the users and presents the changes in an interpretive manner. It is only when the computed differences are presented or displayed in an intuitive and meaningful way that the user can interpret what has changed. Two schemes are presented in this thesis for presentation, namely,

only change approach and the dual frame approach. In the first scheme, only change approach, only changes are displayed in a tabular manner. Dual frame approach has the the old and new versions of the page shown side by side, with the changes highlighted. As the HTML tags are used for presentation and XML tags define content, the presentation of HTML and XML differ. This module is elaborated in chapter 5.

Users have a choice in the medium of notification and the frequency by which they want to get notified of the detected changes. The notification module delivers notifications to the users with a frequency that was opted 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 their 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. This module is explained in detail in chapter 3

CHAPTER 3

NOTIFICATION OF CHANGES IN WEBVIGIL

As the world wide web has been burgeoning at an unbelievable speed, the users' interest have now evolved from viewing of information to selective change monitoring. Users need to be notified at an interval specified by them. This chapter first outlines the sentinel specifications to better understand the requirements of this module. Then it discusses in detail various approaches proposed for notifying the users in a timely manner. Active capability [11, 11] has been very well established in databases. It has been used in a broader context in WebVigiL. This chapter also discusses in detail the use of active rules for interval-based notification.

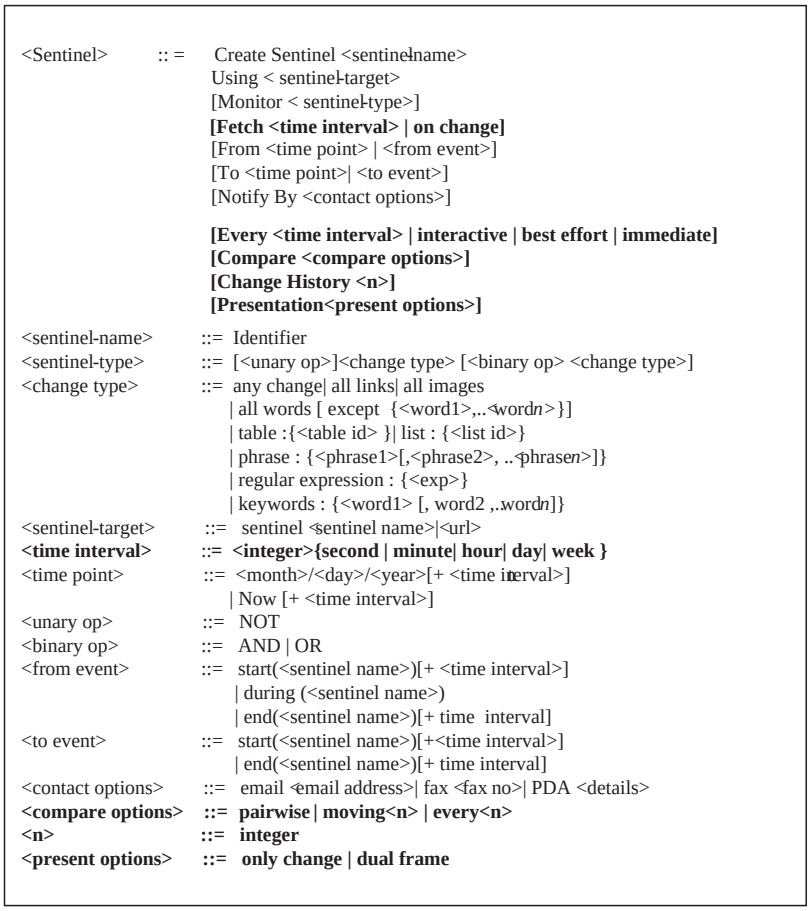
3.1 Sentinel Specification

Users are facilitated with a web user interface to provide the required input to the system. Users have to be notified of the changes detected by the system on the pages of interest. As seen in the Figure 3.1 there are several options for the frequency of notification and the type of notifications [14].

3.2 Frequency of notification

One of the very important criteria of notification is the frequency by which the users want to get notified. The system facilitates the users with several options in the specification which are listed below:

- Immediate
- Best-Effort



<Sentinel> :: = Create Sentinel <sentinel-name>
Using < sentinel-target>
[Monitor < sentinel-type>]
[Fetch <time interval> | on change]
[From <time point> | <from event>]
[To <time point>| <to event>]
[Notify By <contact options>]

[Every <time interval> | interactive | best effort | immediate]
[Compare <compare options>]
[Change History <n>]
[Presentation<present options>]

<sentinel-name> ::= Identifier
<sentinel-type> ::= [<unary op>]<change type> [<binary op> <change type>]
<change type> ::= any change| all links| all images
| all words [except {<word1>,..<word*n*>}]
| table :{<table id> }| list : {<list id>}
| phrase : {<phrase1>[,<phrase2>, ..<phrase*n*>]}
| regular expression : {<exp>}
| keywords : {<word1> [, word2 ,..word*n*]}
<sentinel-target> ::= sentinel <sentinel name>|<url>
<time interval> **::= <integer>{second | minute| hour| day| week }**
<time point> ::= <month>/<day>/<year>[+ <time interval>]
| Now [+ <time interval>]
<unary op> ::= NOT
<binary op> ::= AND | OR
<from event> ::= start(<sentinel name>)[+ <time interval>]
| during (<sentinel name>)
| end(<sentinel name>)[+ time interval]
<to event> ::= start(<sentinel name>)[+<time interval>]
| end(<sentinel name>)[+ time interval]
<contact options> ::= email <email address>| fax <fax no>| PDA <details>
<compare options> **::= pairwise | moving<n> | every<n>**
<n> **::= integer**
<present options> **::= only change | dual frame**

Figure 3.1 Sentinel Specification.

- Interval Based (Use of ECA Paradigm)
- Interactive

From the above list, the initial three types of notifications are push based. WebVigiL system pushes the notifications to the users. But the last is different in which users login to the WebVigiL system and can explore and navigate the changes stored for sentinels defined by them using a user-friendly dashboard. Changes are presented using multiple options for the sentinels whose changes have been detected and stored.

3.2.1 Immediate Notification

In cases where the changes are critical, the user may want to be notified as soon as changes occur on particular pages. In such cases, immediate notifications should be sent to the users. These notifications are delivered at the highest priority by the system. The changes are queued up in two queues, namely, immediate queue and best-effort queue. The changes that have to be notified immediately are queued in the immediate queue. The immediate queue is served prior to the best-effort queue. Changes are picked from the queue, inserted into the database and users are notified immediately. Changes corresponding to interval-based notifications are queued up in immediate queue. They are persisted in the database when they are served by notification module. The later part of this chapter discusses the manner in which interval-based notifications are handled.

An example scenario for specifying a sentinel with immediate notification is as follows:

Create Sentinel S1 ON
http://data.uta.edu/discus/messages/202/203.html
Links Change
From Now To April 8, 2004
Fetch On-Change Compare Pair-wise
Notify me Immediately Via Email

In the above sentinel the user will be notified as soon as the change occurs. Ideally for this mode of notification, there should not be any delay in notification after the change has been detected by the system.

3.2.2 Best-Effort Notification

At times where the change is not critical to the users but the users want to be notified as soon as the page changes, the option of best-effort is preferred. The delivery

semantics of best-effort are similar but subordinate and subsequent to immediate mode delivery. The changes that need to be notified at best-effort are queued up in the best-effort queue. The best-effort queue is served only if the immediate queue is empty. The changes that are picked from the best-effort queue are inserted into the database for persistence before notifications are delivered. Hence it can be seen that there is a subtle difference in the above modes of notification. Best-effort is at a lower priority than immediate.

An example scenario for specifying a sentinel with best-effort notification is as follows:

Create Sentinel S2 ON
http://data.uta.edu/discus/messages/202/203.html
Links Change
From Now To April 8, 2004
Fetch On-Change Compare Pair-wise
Notify me at Best-Effort Via Email

Figure 3.2 shows the flow of data of the changes being stored and the mechanism in which the queues are served.

3.2.3 Interval-Based Notification

Alternatively, frequency of change detection will be very high for web pages that are modified frequently. Since frequent notification of these detected changes will prove to be a bottleneck on the network, it is preferable to send notification periodically. In this mode of notification users are required to specify the frequency at which they want to get notified.

An example of a sentinel specified with interval-based semantics for notification is as follows:

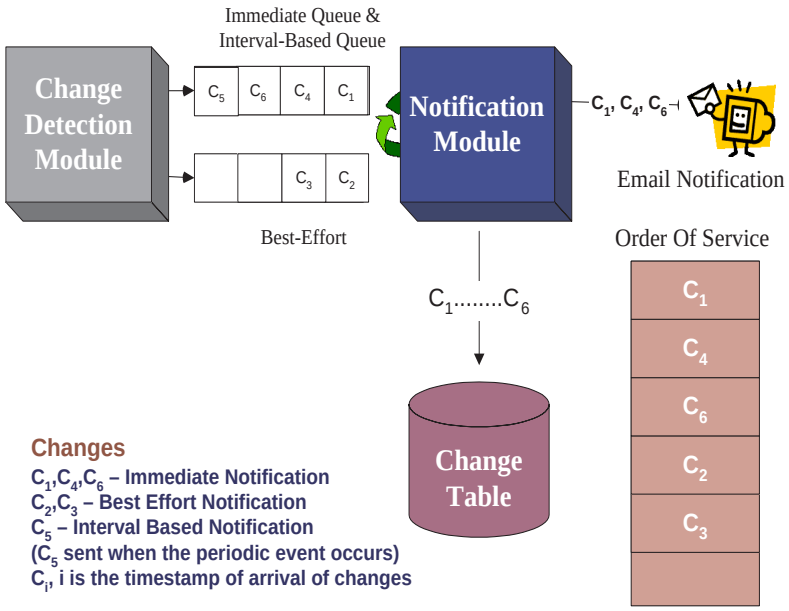


Figure 3.2 Data Flow in Notification.

Create Sentinel S1 ON

http://data.uta.edu/discus/messages/202/203.html

Links Change

From Now To April 8, 2004

Fetch On-Change Compare Pair-wise

Notify me once a day Via Email

In the above example the users are notified once a day with the latest change computed before notification. Although previous n changes (n specified by the user while installation of sentinel) can be viewed by the users by logging into the WebVigiL system and navigating through the dashboard.

Use of ECA Paradigm: As notifications need to be delivered at regular intervals (specified by the user during the installation of the sentinel), a triggering mechanism is needed. We use the active capability to deliver notifications asynchronously. The

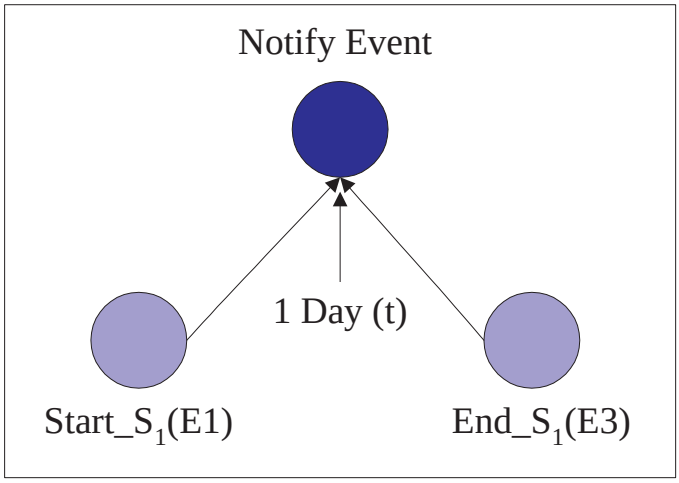


Figure 3.3 Periodic Event for Notification.

paradigm used for notification here is ECA, where (a) E stands for an EVENT of occurrence, (b) C for CONDITION which is evaluated when the event occurs, (c) and A for ACTION is performed if the condition evaluates to true. With the help of periodic events, we trigger notification to the users at their specified intervals. A periodic event [9, 8] is an event that repeats itself with a constant and finite amount of time.

Periodic event is specified as PeriodicEvent (E1, [t], E3) where E1 and E3 are events (or time specifications) that act as an initiator and a terminator, and t is the time interval [8]. Periodic events have been used in WebVigiL for triggering various operations such as fetching, notification and so on. Here E1 and E3 are the start and end events of a sentinel and t is the interval at which the page should be monitored. A separate periodic event is created for notify by the ECA rule generation module in addition to the other events created for the monitoring request. The logical representation of the periodic event (Notify_Event) is shown in Figure 3.4. To actually notify a change, we associate a notify rule with this periodic event. Hence whenever a notify event occurs, the rule associated with it is triggered. This notify rule performs the functionality of the notification; that is, it notifies the page based on changes to the page properties. For sentinels that explicitly

specify the notification interval, we generate a periodic event and associate a notify rule with it. Hence for every sentinel that requires interval-based notification, we generate a unique periodic event and associate a notify rule with it. When the rule is fired, the condition part of the rule checks whether there is a new change detected for the sentinel associated with it since the last notification using a change table. Change Table is a relation in the RDBMS where changes are stored in the form of BLOBS (binary large objects). The number of changes stored by the change table for a particular sentinel is a parameter specified by the user while installing a sentinel. A flag is raised in the change table when a new change is inserted. So in the condition part of notify rule, the flag is checked for insertion of a change. If there exists a new change inserted into the database for that sentinel, the action performed by the notify rule is to notify the user that has registered the corresponding sentinel.

Consider the scenario where sentinel s1 specifies the notify time as once a day. The creation of the periodic event generated for s1 is as follows:

$$\text{Event Notify\_Event\_S}_1 = \text{createPeriodicEvent (Start\_S}_1\text{, t, End\_S}_1\text{)}$$

where Start_S_1 and End_S_1 are the start and end events of sentinel S1. The event Start_S_1 is a primitive event that initiates the periodic event Notify_Event_S_1. For every interval t (1 day) the periodic event is raised until event End_S_1 occurs which is again a primitive event. When the periodic event is raised, the notify rule associated with it is fired to start notification for the corresponding sentinel.

3.2.4 Interactive Notification

WebVigiL provides a facility to view the information of their sentinels and changes maintained by the system. A user *dashboard* is provided for that purpose. Interactive notification is a navigational style retrieval where the users visit the WebVigiL dashboard to retrieve the detected changes at their convenience. Through the WebVigiL dashboard

the users can view and query the changes generated by their sentinels. The users are given an option to provide the number of previous changes they want the system to maintain. The maintenance of changes entails that versions involved in the changes be stored and managed properly in order to compute and render the changes at user request. This has an impact on the storage of versions and the storage overhead can be significant. It requires the version management to include the above as part of the deletion of versions.

An example scenario for specifying a sentinel with interactive notification is as follows:

Create Sentinel S4 ON

http://data.uta.edu/discus/messages/202/203.html

Links Change

From Now + 2 days To End(S1) + 1 month

Fetch On-Change Compare Pair-wise

Notify me at Interactive and store last 5 changes

3.3 Media of Notification

The media for notification listed in the specifications are as follows:

- Email
- Fax
- PDA

Currently the system supports Email. We plan to extend the system to handle the remaining two media of notifications.

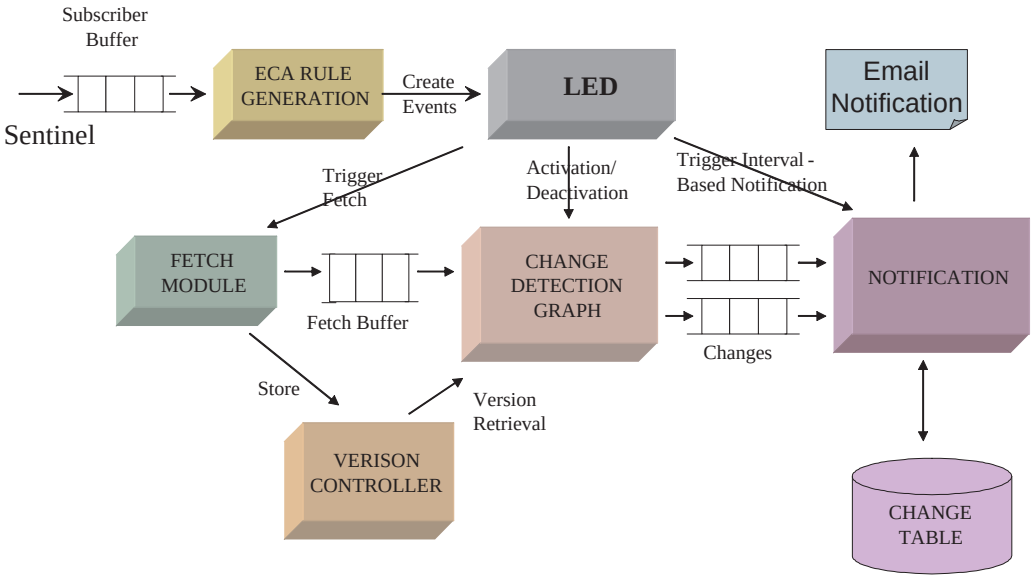


Figure 3.4 Complete Flow of Notification.

3.4 Notification content

Notification content has to be concise and lucid. The notification report contains the following basic information:

- User specified name of the sentinel
- Target URL of the sentinel for which change has been detected.
- User specified type of change, such as “any change”, “all words” etc.
- Link to a page where the changes are rendered at run time.

The size of the notification report is kept small in order to satisfy user requisites taking into account the network quality of service requirements.

3.5 Summary

A complete flow of data for notification is as shown in Figure 3.4. When the users provide the input, notify events are generated only if interval-based notification is preferred. When changes for sentinels that require interval-based notification are detected, they are inserted into the change table which is discussed earlier in the chapter. As the notify event and corresponding rule fires to notify the users. If the sentinels require immediate or best-effort notification, the changes are inserted into corresponding queues and the notification module serves the queues and notifies the users and stores the changes into the change table.

CHAPTER 4

PERSISTENCE AND VERSION MANAGEMENT

As change detection requires two versions of a page for comparison, and the same page is retrieved multiple times over a period of time, there is a need for storing, retrieving, and managing persistent versions of web pages of interest. This chapter presents an efficient mechanism to store, manage, and provide the appropriate pages as needed to perform change detection. The chapter also discusses in detail, the technique used to discard the versions when they are no longer needed. The functionalities required of the version manager are as follows:

- Manage the storage of retrieved pages.
- Manage and provide the pages needed by the *Change Detection Module*.
- Provide interface to *Fetch* and *Presentation* modules.
- Reduce the number of unnecessary fetches by checking the meta-data of the stored pages.
- Maintain a mapping of URL names to a directory structure for the easy retrieval from the physical storage.
- Delete pages that are no longer needed.

The primary purpose of the repository service is to reduce the number of network connections to the remote web server, thereby reducing network traffic. It reduces network traffic and latency for obtaining the web page as the it avoids redundant fetches by checking the meta-data which is discussed in the following section. The quality of service for the repository service includes managing multiple versions of pages without excessive storage overhead.

4.1 Issues

The issues that are addressed in the design of version management module are as follows:

- Determine whether to fetch a page using meta-data
- Determine when to delete a page
- Efficient mapping of URLs to a directory structure.
- Provide pages to the change detection module
- Efficiency

The subsequent sections discuss how we handled the issues listed above

4.2 Determine whether to fetch a page

A page is fetched only when there is a change to the page after the previous fetch. The criterion that decides the page to be fetched is a change in the meta-data of the page. The meta-data of the page includes page properties such as:

- Last Modified Timestamp (LMT)
- Checksum

A page is fetched when the page properties change. So, before fetching the entire page (i.e., its contents), the meta-data of the page is fetched. If there is a change in the meta-data (Last Modified Timestamp). Based on the nature (static/dynamic) of the page being monitored, the complete set or subset of the meta-data is used to evaluate the change. For static pages, HTTP HEAD request is used to obtain the meta-data of the page. A page is fetched if there is a change in the timestamp or there is an increase of decrease in the size of the page. In cases where the time stamp is modified, but the page size remains the same, HTTP GET request is used to retrieve the page and the checksum of the page is calculated. For pages that are not provided with last modified

timestamp such as dynamically generated pages or cases where previous attempts to retrieve page properties failed, HTTP GET request is used to retrieve the page. Even in this case, whether the change should be computed is based on the checksum value. *Version Manager* determines and signals the *Fetch Module* whether the page needs to be fetched. The version manager maintains run-time data-structures which are shared by several threads executing the rules for fetching a page. They store certain information about the version of the page previously fetched. There are mainly three hash-tables in the version manager that store the data depending on the fetch rules. They are, *LatestHash*, *BestEffort Hash* and *FixedInterval Hash*. The information of the version of the page fetched is maintained in the *Version Object*. The Version object contains the following information:

- Version Number
- Last Modified Timestamp LMT
- Checksum
- URLMapping

Every URL is mapped to a directory on the physical storage which we will discuss in the subsequent sections. For now we can look at it as every URL having a unique mapping which represents the URL throughout the system. The LatestHash is hashed on URL (URL mapping). The value in the hash-table bucket of the LatestHash is the Version object of the latest version of the page fetched. The BestEffort hash also has URL(URL mapping) as its key and a list of Version objects is stored as a value in the bucket for a particular URL. The version pertaining to a fetch when best-effort rule [15] is triggered is inserted into the list that is stored for the corresponding URL. The FixedInterval hash is hashed on SID(sentinel Id) which is a unique Id given to every sentinel that is registered with the system. It contains a list of versions fetched for a particular SID (or URL) as its value when the fixed-interval rule for fetch is fired. There can be Version

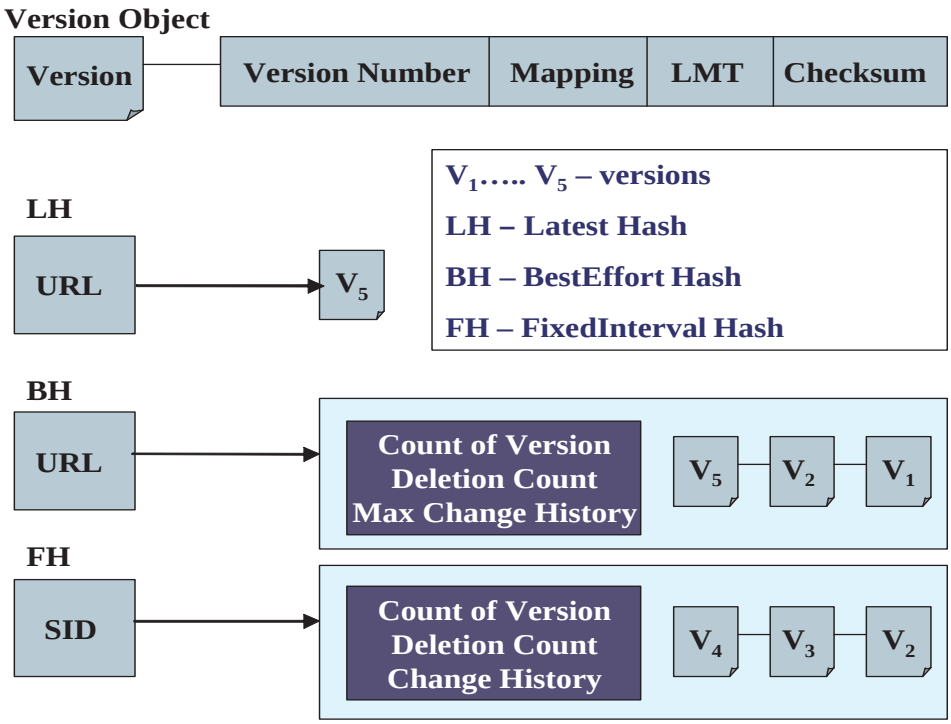


Figure 4.1 Version Manager Runtime Data-structures.

objects that are common to both the BestEffort hash and FixedInterval hash as both the rules can be fired at the same time for the same URL. This arises due to the fact that there are two or more sentinels using the same version of the page. As seen in the Figure 4.1 BestEffort Hash and FixedInterval Hash also contain a Deletion Count, Count of Versions and change history which will be discussed later in the chapter. The mechanism by which the version controller decides and signals the *Fetch module* if a fetch is required is discussed in detail below:

When a fetch rule is fired the LatestHash is checked for the URL. If it does not contain the URL, it indicates that the page is to be fetched for the first time in the system. So the page is fetched and stored and an entry is made in the LatestHash. If the

LatestHash has a entry for the URL, it means that the page has been fetched previously. The Version Object stored in the LatestHash will have the properties of the version of the page fetched the last time. The meta-data of the current fetch is compared with that of the previous version. If there is a change, the fetch module is signalled to fetch the page. When the page is fetched an entry for the latest version is made in the LatestHash and depending on the type of fetch (BestEffort or FixedInterval) the version information is also added in the list of Version objects stored in the respective hash tables. If the URL is registered with LatestHash and there is no change in the meta-data since the last fetch, the respective hash table for the type of rule fired is checked for the URL. For example, if the previous fetch was for best-effort rule on a particular URL then BestEffort hash will contain the information of the previous version of the page. The meta-data of the current version and that of the previous version in the BestEffort hash are compared and if the information is not the same, then the details of the current page are attached to the list of Version objects stored in the BestEffort hash. If the previous fetch of that page was for fixed-interval rule, and as the page has not changed after that, the BestEffort hash is populated with the same version information about the page as the Fixedinterval hash contains and a fetch is saved.

There can be several sentinels on the same page with best-effort rule for fetch. As sentinels with best-effort rule are grouped, once a page is fetched, the version is provided to all the sentinels with best-effort fetch rule interested on that page for change detection thereby saving lots of unnecessary fetches. The page is fetched just once for all those sentinels. This version is shared by all sentinels that are on the same page but have different best-effort fetch rules associated with them. In addition, there can be a sentinels with fixed-interval fetch rule also sharing the same version for change detection.

As we can see in the Figure 4.1, the LatestHash contains V5 which is the most recent version of the page fetched. It can be seen that the newer versions coming in

are stored at the head of the list which is a benefit when the list is traversed to provide a version required by the change detection. V1 and V2 are fetched by the best-effort rule for a particular URL. Version V2 is fetched only once because the next time it is requested for fetch, the version manager indicates the fetch module that there is a copy of the the version in the page repository and the fetch is not needed. So V2 is shared between a sentinel with best-effort fetch rule and one with a Fixed-Interval fetch rule. V3 and V4 are fetched for a sentinel with fixed-interval fetch rule. Thus we can see that by storing some meta-data information the version controller can save some fetches which are not required.

The version manager provides the versions stored to the change detection module as and when needed. The change detection module provides the URL, the sentinel details for which the change detection is taking place. In addition it also gives the reference version number and the offset of the previous version it requires for change detection. The version manager finds the version needed using the offset and by going through the lists in the respective hash-tables and provides the required version to the change detection module. For example, if the change detection for a version fetched by best-effort rule, the reference version number is V5 and the offset is 2. The version that is provided by the version controller as seen in the list for BestEffort hash in Figure 4.1 is V1 which is two versions older than V5 in that list.

4.3 Deletion of a page

As the page repository has large conglomerate of pages, there is a large storage overhead if pages that are no longer needed are not deleted. Hence there is a critical need for purging the pages when no longer required. Deletion of pages not only need to take into account the compare option (indicating the versions used of comparison by several sentinels on the same page), but also the number of changes that need to be kept

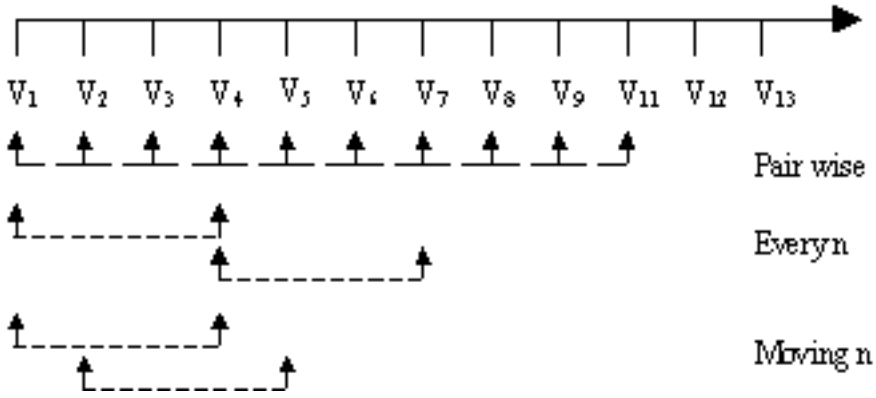


Figure 4.2 Compare Options.

for an interactive notification. We propose several approaches which are discussed below and finally narrow them down to an approach that is better computationally and is on the conservative side to make sure needed pages are not deleted.

4.3.1 Parameters for deleting a page

There are two factors that play a vital role in deciding if a version of a page is no longer required. The factors are:

Compare Option: As shown in Figure 4.2 it is the relative reference number for the version of the page specified by the user to be used for change detection. For example if the user specifies the comparison option as every(3) as explained in chapter 2, the previous three versions associated with that sentinel cannot be deleted. There is a possibility that the coming version may require one or all of them for change detection depending on the compare option.

Change History: The number of changes that the user wants the system to persist. For example, the user wants the last 5 changes to be stored so that they can be retrieved using the dashboard. So the versions associated with those last 5 changes

cannot be discarded as they will be needed by the presentation module to render the changes when needed.

4.3.2 Naïve Approach

The approach that was considered initially was persisting the changes in a table in DBMS where each change for each sentinel is a row in the table. As the number of rows would be the cross product of the changes detected for each sentinel and the number of active sentinels in the system, the size of the table was suspected to be huge. The details stored in the table would be:

- Sid
- Change Object
- URLmapping
- Time-stamp of detection of the change
- The version numbers of the two versions of the page involved in change detection.

Whenever the notification module has to insert the change for a particular sentinel in the change table, the number of rows in the change table for that sentinel have to be summed up. If it exceeds the change history, the row with the lowest time-stamp has to be searched and deleted. JDBC calls are blocking calls and performing all the above activities requires many SQL queries to be executed by the DBMS, so the notification thread which inserts the changes into the data-base is blocked until the DBMS completes all the SQL operations mentioned above. When deletion is triggered for a particular URL, the change table would be accessed again and a *list of versions* to be retained for that URL are congregated. The versions other than those in the list are searched for and are deleted in the physical storage as well as the runtime data-structures. It required going though a list on the BestEffort hash for that URL and as many lists in FixedIntervalHash as many sentinels are interested in the URL to check each version that can be deleted.

This can be done by comparing the version numbers in the lists of both the hash tables with that of *list of versions* which is formed by looking up the database for the required versions. It is computationally very expensive to perform several operation as discussed above for a single deletion sweep. So though it will give very accurate results other computationally less expensive options were considered.

4.3.3 Present Approach

One of the primary problems that needed attention in deleting the pages was to quantify an interval at which the version controller checks for versions that are no longer needed. It cannot be very small because that adds considerable computational overhead on the system whereas it cannot be as large as the system has to store a large number of unnecessary versions. When the number of versions for a particular URL exceeds a specific number, deletion is triggered. The number mainly depends on the quantities described above. The BestEffort hash stores in the attribute *Deletion Count*, the number of versions for the page it might need in the future considering the compare options. The Deletion count is updated every time a new sentinel is inserted and a page is fetched for the same. It is updated only if the number of versions to be retained has be greater than what it currently retains. The *Maximum Change History* in the BestEffort hash is the maximum number of changes associated with the page regardless of any sentinel, that have to be stored. It is checked every time a sentinel is placed on the system and modified if the change history is greater than the current value. The FixedInterval hash stores information based on each sentinel. Hence the *Deletion Count* is put in the first time sentinel is installed, which is based on the compare options specified by the user for that particular sentinel. The *Change History* in FixedInterval hash is stored once the hash table is populated with the details for the sentinel.

If the number of versions for a URL exceeds the product of *Deletion Count* and *Max change History* for BestEffort hash there is a possibility that some versions are not required anymore and can be purged. For FixedInterval hash when the number of versions are greater than the product of *Deletion Count* and *Change History*, some versions can be discarded. Deletion is triggered at *(n * Deletion Count * Max Change History or Change History)*, where n is a positive integer. The greater the value of n, the more version get deleted in a single deletion sweep. n is specified as a configuration parameter, hence can be test to different values. For greater values of n, deletion will be triggered less frequently, but at the same time there will a large storage of unnecessary versions.

For example, if in the BestEffort hash the deletion count is 4 which means that one of the sentinels using best-effort fetch rule has the compare options where a page four versions older than the current version is to be used for change detection with the current version. And the maximum change history for all the sentinels is 3. Assume the value of n is 1. So the probability that some versions can be discarded when the number of versions for that URL exceeds *n(1) * Deletion Count (4) * Max Change History (3)* is high and the system checks for deletion at this point.

The number of changes that the user wants the system to maintain are stored in the KnowledgeBase in the form of BLOB objects in the change table. The change table stores various parameters shown below:

- Sentinel ID
- URL (URL mapping)
- A list of changes stored as a BLOB object
- Fetch type
- Minimum version number

- A flag used by the notification module to check is a new change has been inserted for that sentinel after the last notification

The table contains one row per sentinel. So the size of the table will be as many number of rows as the number of sentinels in the system (active or inactive). The minimum version number is the oldest version number that is required for rendering any change that is persisted in the database. Changes older than the minimum version number are discarded from the table. The minimum version needed for presenting the changes is updated every time a change is discarded.

When deletion is triggered the version controller checks the change table and computes the minimum version number needed and both the hash tables. That is, *BestEffort hash* and *FixedInterval hash* are checked to see if they have versions older than the minimum version needed for that URL. In addition, deletion count in both the hash-tables is also considered. The versions that are older than the deletion count and minimum version needed for presentation are deleted from the system. The advantage of this approach is that when there are many versions that can be deleted, they all are deleted when a check for the deletion is done. The batch processing of deleting the versions saves a lot of computation. An example showing how versions are stored and discarded is explained at the end of this chapter.

4.3.4 Comparison of Approaches

Naïve Approach:

- Computationally very intensive
- when not needed, a page is deleted as soon as possible
- Deletes pages one at a time

Present Approach:

- Requires less computation

- conservative approach
- Deletes pages in a batch.

When the approaches are compared, the Present approach is preferred. The overhead of computations involved in deleting pages using the naïve approach is more than that that of the present approach. A trade off with the present approach is that it is a bit conservative in deleting the pages as compared to the naive approach, but less computationally intensive.

4.4 Naming Convention for Storing Pages in Cache

When a page is fetched it is stored on a physical storage. In order to detect changes to a particular page, versions of that page have to be stored in the cache. To maintain these versions in the cache, each URL has to be mapped to a unique directory. The complete URL cannot be used as a directory name since the length of the URL is very large in many cases. The approach used by WebVigiL [12] to map the URL to a directory is as follows: The URL is mapped to a directory structure. For example, for the URL "a/b/c/x.htm", the directory structure can be "a/b/c/x.htm". But, in case of dynamic pages the filename (i.e., "x.htm") can be very large. There are restrictions by the underlying system on the length of the directory name. Hence directories for dynamic pages cannot be created. Therefore, a hash-based approach is used to generate a unique mapping for the directory as well as file name. Thus, the directory structure is a concatenation of URL mapping up to the filename with the file mapping generated. For the above example "a/b/c" is mapped to a unique mapping for the directory "D1" and the file "x.htm" is mapped to F1 and all versions of the page are stored under the directory "D1F1".

Two hash-tables – one each directory mapping and file mapping are maintained by the system to store and provide the URL mapping at runtime. Figure 4.3 shows

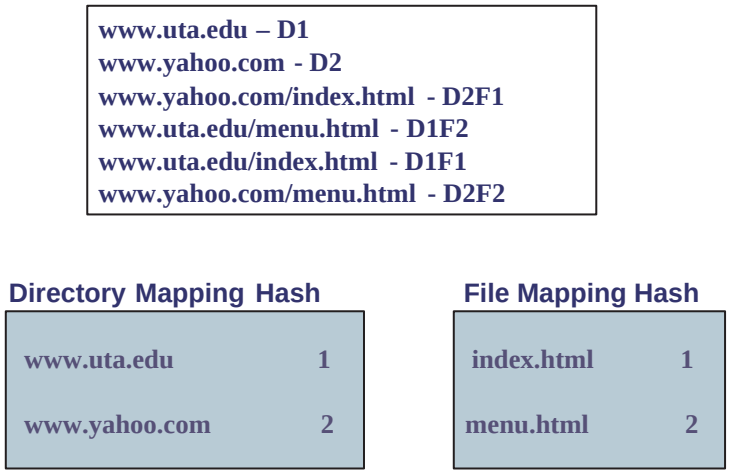


Figure 4.3 Mapping of URL names to a directory structure.

examples of how a URL is mapped to to a directory structure and the information of mapping maintained by the version manager.

As we can see, for the URL "www.yahoo.com/index.html", complete string before the filename "index.html" is mapped to a directory "D2" and the filename is mapped to "F1". Both are concatenated and the complete mapping for the URL is"D2F1".

4.5 An Example to illustrate the functionality of version manager

Table 4.1 shows the details of four sentinels set on the same page with different fetch frequency and compare options.

In this section, we discuss the complete functionality of the version manager using a few examples. As shown in the Figure 4.4, we have stored versions [1-13] of "www.uta.edu" fetched for sentinels [1-4]. The URL "www.uta.edu" is mapped to "D1"

Table 4.1 Example Sentinels for Version Manager

SENTINEL	URL	FETCH TYPE	COMPARE	# OF CHANGES
S1	www.uta.edu	BE	PAIRWISE	2
S2	www.uta.edu	BE	EVERY (3)	2
S3	www.uta.edu	FI	PAIRWISE	2
S4	www.uta.edu	FI	MOVING(3)	2

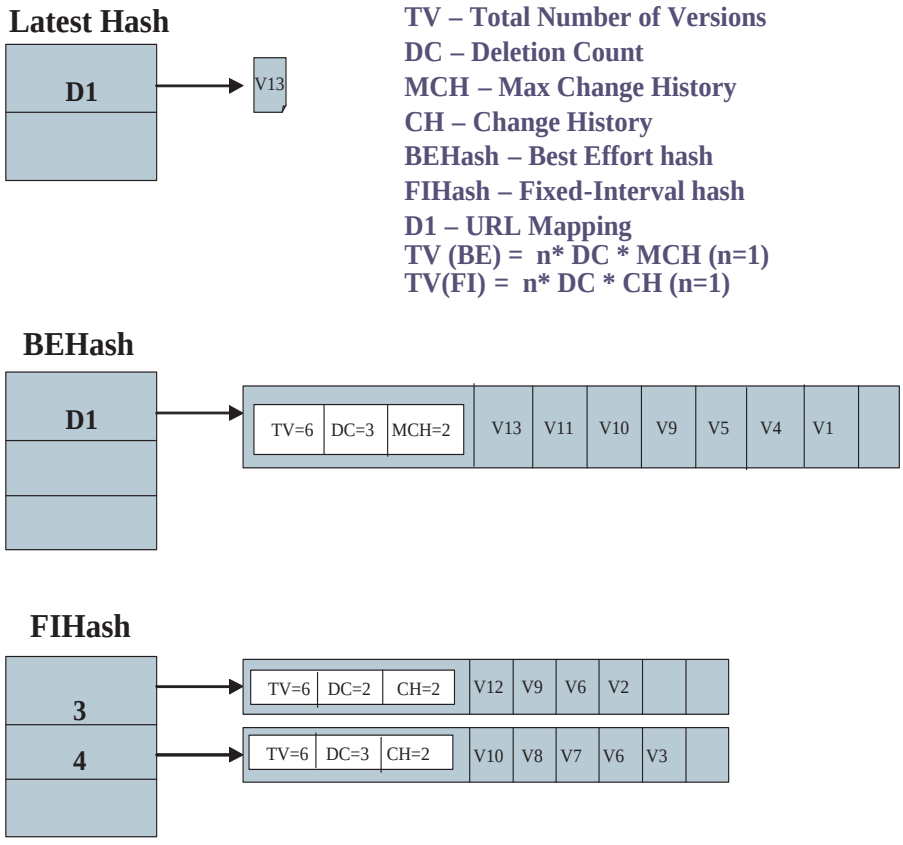


Figure 4.4 Example of Storage and Deletion in Version Manager.

directory. When a page fetch is initiated, the version manager checks for the existence of the latest version of the page in the LatestHash and if present, the latest version of the page in the cache is returned. In the case of static pages, if the LMT of the page is equal to the LMT of the cached copy it is not fetched. In the case of a dynamic page, the page is fetched and the checksum of old version in the cache is compared with the checksum of the fetched page. If the checksums are equal then the page is not stored in the cache to avoid duplication of the versions. For example version V9 is fetched for a best-effort sentinel. When fixed interval rule is fired the version page is already the latest version that exists in the cache so the details of the version are inserted into the Fixed-Interval hash and the page is not fetched again. Thus unnecessary fetch of a page is avoided by maintaining some run-time information. In case of a cache miss, the repository service requests that the page be fetched from the appropriate web server.

Table 4.2 Example of Change Table

SID	URL	LIST OF CHANGES	MIN VERSION	FETCH TYPE	FLAG
S1	D1	(V11,V13),(V10,V11)	V10	BE	T
S2	D1	(V11,V13),(V9,V10)	V9	BE	T
S3	D1	(V9,V12),(V6,V9)	V6	FI	T
S4	D1	(V8,V110),(V7,V8)	V7	FI	T

Assume that each comparison generated a change. Now as all the sentinels need the last two changes to be persisted by the system, the change table will be as shown in the Table 4.2. The *MIN VERSION* column is the minimum version needed by the oldest change in the *LIST OF CHANGES* column. For example, For Sentinel S1, the oldest changes persisted is between versions V10 and V11, so the minimum version needed for rendering the changes for Sentinel S1 at this point of time is V10. Once a change is

removed from the *LIST OF CHANGES*, the value of the column *MIN VERSION* is updated.

As discussed earlier, deletion will be triggered when the total number of versions in the list stored in the BestEffort hash or FixedInterval hash exceeds a certain value. This value depends on the *Deletion Count* and *Change History* as mentioned in section 4.3. The deletion count for the URL "www.uta.edu", which is mapped as "D1", in the BestEffort hash is the maximum of the two compare options of the two sentinels having best-effort fetch frequency. It is updated if a sentinel with a greater compare option for the same URL has entered the system. The deletion count for each sentinel in FixedInterval hash is set according to the compare options of the corresponding sentinel. The same argument applies to the value of change history as the deletion count for both the hash-tables. As shown in Figure 4.4 for the list in the BestEffort hash, the deletion count is 3, the max change history is 2. The value of n is assumed to be 1 for the above example. It can be changed as it is a configuration parameter. So if the total number of versions exceed 6 (n * deletion count * max change history). When version V13 comes in the number of versions in the list is greater than 6 so version manager checks if any versions can be deleted. The minimum of the MIN VERSION column in change table is V6. Therefore V6 is the oldest version needed for presentation by any sentinel on URL "www.uta.edu". The deletion count for the list in BestEffort hash is 3 so the oldest version required for future comparison for any best-effort sentinel is V10. Similarly V7 and V9 are the oldest versions required by sentinel 3 and sentinel 4 respectively in the FixedInterval hash. Considering all the factors, we conclude that the versions older than V6 are no longer required and are deleted from the physical storage. The run-time information of deleted versions from the hash-tables is erased as well. Versions V1-V5 were deleted at once. We process the versions for deletion in a batch to avoid the overhead

that is involved in deletion getting fired very frequently. The examples in the current section gives a clear idea of the functionality of the version manager.

4.6 Summary

Thus this chapter contains a detailed description of various functionalities of version manager which is an essential part of WebVigiL system. Version manager stores provides the versions of the pages required by different modules of the system multiple number of times over a period of time. It avoids a page to be fetched unnecessarily by storing some meta-data of the page and indicating the fetch module to fetch only if the meta-data has changed. It also incorporates a mechanism to delete the versions of the page when the version is not required by the change detection module as well as the presentation module.

CHAPTER 5

PRESENTATION OF CHANGES

The final and an important aspect of change detection in WebVigiL is the presentation of detected changes in an easy-to-understand manner. Therefore, computing and displaying the detected differences in a meaningful manner is very important. The users may be interested merely in the changes on the page of interest or they may be interested in understanding the changes along with the context to interpret them better. Hence WebVigiL facilitates the users with two types of presentation, namely "Change-Only Approach" and 'Dual-Frame Approach". The users are notified with an email of the changes detected for their sentinels. The users can specify from the 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 this an option to choose "interactive" notification is provided to the users as discussed in chapter 3. Also, the users may want to lookup previous changes at any point in time. The users are provided with a dashboard which is an interface for the users to locate the last n changes. n is number of changes detected and maintained as part of the history of changes specified by the users at the time of input.

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. As for example as shown in Figure 5.1 the tags in XML define the content as Author/Section/Book but the tags such as table/tr/td are used for presenting the data in HTML. Unlike HTML, the element names in XML have no presentation semantics but instead define the content. Hence the change presentation of HTML and

XML

```
<Section>Children
        <Author>JRR Tolkien</Author>
        <Book>Lord of the Rings</Book>
</Section>
```

HTML

```
<table>
   <tr>
      <td>Section</td>
   </tr>
   <tr><td>Author</td>
      <td>Book</td>
   </tr>
   <tr>
      <td>JRR Tolkien</td>
      <td>Lord of the Rings</td>
    </tr>
</table>
```

Figure 5.1 Differences in HTML and XML.

XML is handled in a slightly different manner. In the rest of the chapter contains a detailed discussion on the presentation schemes used in WebVigiL for change presentation in HTML and XML separately.

5.1 Content in Presentation

The presentation of the detected differences should be concise and lucid to the users. Users should be able to clearly perceive the computed differences in the context of his/her predefined specification. The presentation contains the following basic information:

- URL for which the change detection is invoked
- The type of change (for e.g., Keywords, Phrases)
- Notification time
- A Link for viewing the type of presentation selected (Only Change/Dual Frame)

The content of the notification should be as minimum as possible to satisfy the network quality of service requirements.

5.2 Change Presentation for HTML Pages

The changes detected currently are stored in a change repository. The change repository has been discussed in chapter 4. 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 gives elaborates on the two schemes used for presenting changes in HTML pages.

5.2.1 Changes-only Approach

This approach gives users a brief account of the changes that have occurred in the pages of interest. For large web-pages, if the user merely wants to know of the changes and not the context with respect to the changes, this approach is very useful. In this approach, changes, not the entire page content, are displayed in an HTML file using a tabular representation along with the type (insert/delete/update). 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. For phrases as the HTML change detection in WebVigiL detects updates to a phrase, the table shows the phrase as well as if it is an insert or a delete. The table also contains a count of occurrences of the content before and after the change as shown in the Figure 5.2 as OldCount and NewCount. For example the keyword "CSE5324" occurred 2 times in the old page and one of the occurrences has been deleted so the new page contains only one occurrence of "CSE5324"

The tabular presentation for a sentinel with a composite change type (links AND images AND keywords (CSE1320,CSE5324) AND PHRASES (ALGORITHMS & DATASTRUCTURES, OPERATING SYSTEMS 1)) is shown in Figure 5.2. The first table in the picture displays the changes in links and images for the sentinel and the second table

Entry	OldCount	NewCount	Change
"/uta/feature-html/multicultural.html"	0	1	Insert
"/uta/feature-html/experientialed.html"	2	1	Delete
"/uta/feature-html/images/cultural.gif"	0	1	Insert
"/uta/feature-html/images/conted.gif	1	0	Delete

Entry	OldCount	NewCount	Change
CSE1320	0	1	Insert
CSE5324	2	1	Delete
ALGORITHMS & DATASTRUCTURES	0	1	Insert
DATABASE SYSTEMS 1	1	0	Delete

Figure 5.2 Tabular Presentation of changes in HTML Pages.

shows changes in keywords and phrases. 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 ths approach is mininmal which is appropriate for devices with less storage and when the bandwidth is small.

5.2.2 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 for interpretation as the contents of the page along with the changes highlighted are displayed. This presentation may be overwhelming when the number of changes are large.

In WebVigiL, the changes are stored in lists which are in turn stored as BLOBs in the in the change table. Whenever the user clicks on the link provided to him in

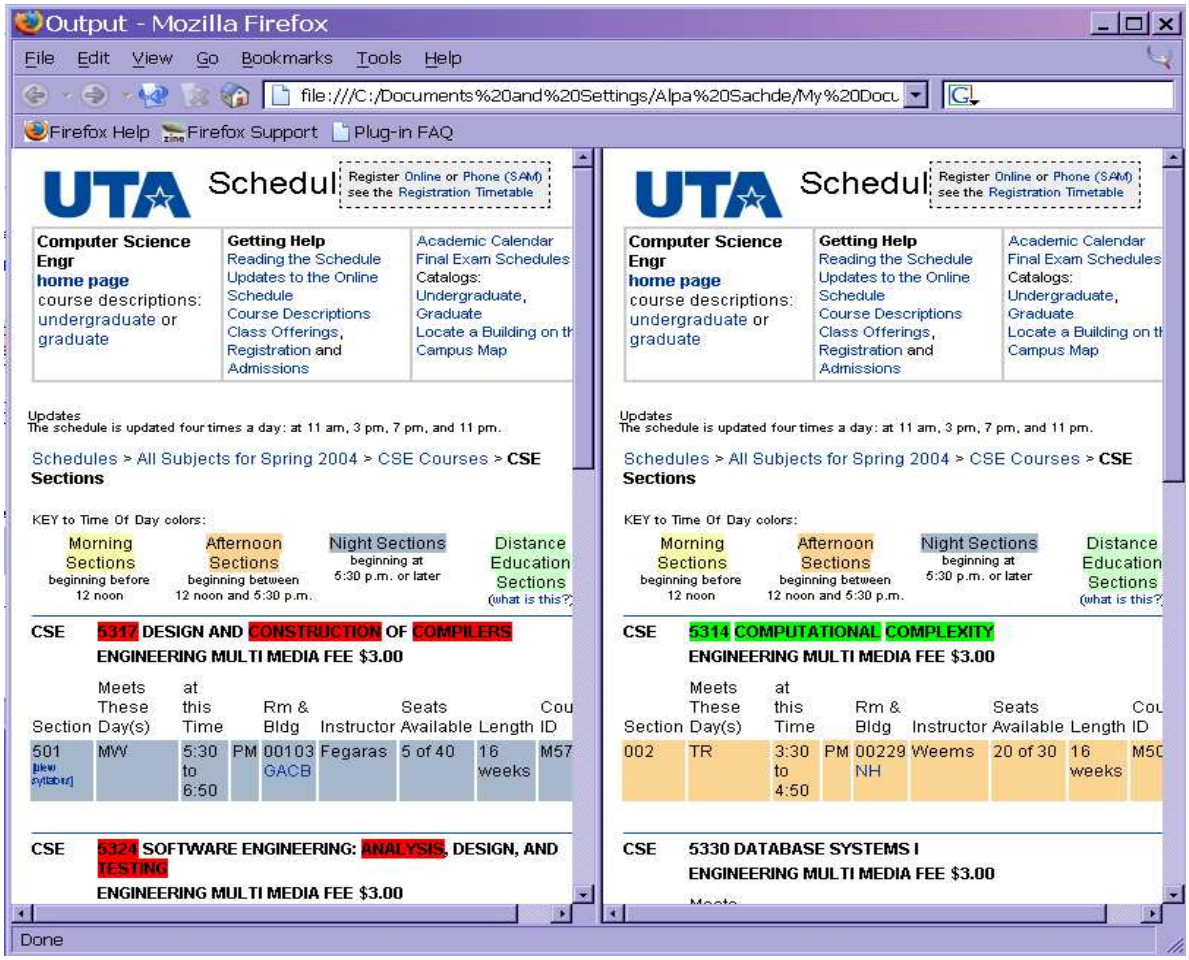


Figure 5.3 Dual Frame Presentation of changes in HTML Pages.

the email or the link for a particular change which appears on the dash-board, the presentation is computed at that point. When a request for dual frame presentation is given to the WebVigiL server, it extracts the corresponding list of changes for the selected sentinel from the database, and retrieves the required versions involved in the detection of that change. The complete page need to be traversed to find the changes identified by the change detection algorithm along with some presentation tags in order to differentiate the changes that are added. For example as shown in Figure 5.3 the keywords "5317","5324","COMPILERS","ANALYSIS" & "TESTING" are deleted and

are highlighted in red in the frame on the left of the page. In addition, the keywords such as “5314”,“COMPUTATIONAL” & ”COMPLEXITY” are inserted and are highlighted in green in the frame on the right of the page. The content of a page that has been deleted is highlighted on the older version of the page. Similarly the content that is added to the page is highlighted on the newer version of the page. Both the highlighted versions of the page are placed in two frames side-by-side. The presentation is supported for LINKS, KEYWORDS AND PHRASES as of now. Currently we do not support the presentation of IMAGES due to certain limitations. For example, images associated with each version need to be fetched and stored. The system currently does not support fetching, storing and managing images within web pages.

5.3 Change Presentation for XML Pages

XML does not contain presentation tags whereas in HTML the data is embedded in the presentation tags. Unlike HTML, the element names in XML have no presentation semantics but instead define the content. HTML browsers are typically hard-coded. Although some browsers format using Cascading Style Sheets (CSS), they still contain hard-coded conventions for documents which do not provide a style-sheet. For XML pages, unless associated with a presentation style, the web browsers would display the document in its source format, which is difficult for a user to understand. Hence, the presentation of an XML document is dependent on a style-sheet.

For presentation, XML data can be stored inside HTML pages as “Data Islands”, where HTML is used only for formatting and displaying the data. The standard style-sheet language for XML documents is the Extendible Stylesheet Language (XSL). It provides a comprehensive model and vocabulary for writing the style-sheets in XML syntax. Stylesheets are used to express how the content of XML should be presented.

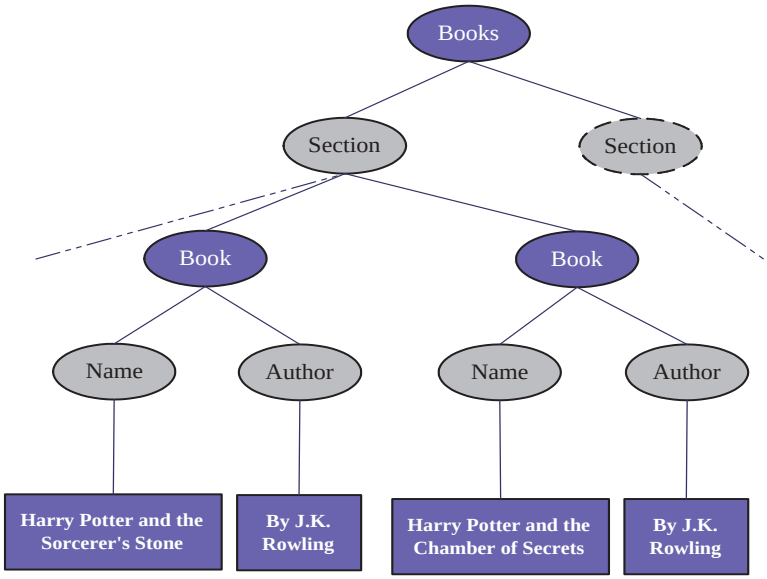


Figure 5.4 Example of Ordered Labeled XML tree.

XSLT (XSL Transformation) [16] accepts the XML document and the XSL for that document and presents it based on the intention of the designer of that style-sheet.

An XML document fetched from the Internet for monitoring 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 the XML document. In such cases, we would have to create a new style-sheet for presentation. Even if the style-sheet is available, we need to modify the style-sheet to highlight the detected changes.

As an example, in Figure 5.4, the node "J. K. Rowling" appears twice for the same context. If only one of the nodes results in either an insert/delete, it would be difficult to identify and highlight that particular node without modifying the existing structure.

At present, two schemes are used to present the changes for a meaningful interpretation by the user. The schemes along with their advantages and disadvantages are discussed below:

```
<? xml version = “1.0”?>
<Electronics Products>
  <New >
     <Item> XY DVD player <\Item>
     <Price currency = dollar>150.85</Price>
  <\New>
  <Discount >
     <Item> TY CD player <\Item>
     <Price currency = dollar> 50.00</Price>
  </Discount >
</Electronics Products>
```

```
<? xml version = “1.0”?>
<Electronics Products>
  <New >
     <Item> Sony CD player  <\Item>
     <Price currency = dollar>200</Price>
  <\New>
  <Discount >
     <Item> Toshiba 21" TV<\Item>
     <Price currency = dollar> 250.00</Price>
  </Discount >
</Electronics Products>
```

Figure 5.5 Old & New Versions of an XML file for Tabular Presentation.

5.3.1 Change-Only Approach

In this approach only the changes are presented. The traditional method is a tabular structure [17] with the types of changes (insert/delete/move) as different columns of the table. Here we only need one style-sheet for any XML document as the changes just need to be embedded into a table in HTML file. The approach is advantageous for documents with large size and large number of changes. That is, if the number of changes is large, Change-only approach is better. But it is difficult for the user to decipher the changes, as these changes are not be shown relative to the original document. This mechanism is also be useful in the future when the notification needs to be sent on a hand-held device where there is a constraint on the amount of data transmitted over limited bandwidth.

As mentioned in the previous sections, the changes are stored as BLOBs in the change table. The lists of changes stored in the BLOB are retrieved in this approach and are inserted into a table in HTML for display as shown in Figure 5.2. Files associated with the changes presented in Figure 5.2 are shown in Figure 5.5. The keywords of interest are “TV” and “DVD”. As it can be seen that the item “XY DVD Player” is

Entry	OldCount	NewCount	Change	Signature (Path)
TV	0	1	Insert	Electronic/Products/Discount/Item
DVD	1	0	Delete	Electronic/Products/New/Item

Figure 5.6 Tabular Presentation of changes in XML Pages.

deleted from the new items in the new version of the file and "Toshiba 21" TV" is added in the discounted items in the new file.

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.

5.3.2 Dual-Frame Approach

In this approach [17] 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 our own style-sheets or embed them in HTML as data islands for us to highlight or strike out the changed contents. The document presented may not have the same look and feel as the original document. This approach is difficult to implement but visually appealing from the user point of view. The Dual-frame approach can be used for keywords and phrases, as the number of changes will be less. This is a viable approach for displaying changes on the WebVigiL dashboard but the constraint of limited bandwidth will make it less feasible on hand-held devices that are planned as a of medium of notification in future. XML is a semi-structured language and can be represented in the form of DOM a tree. In this presentation, we convert the XML into a DOM tree and traverse each node. In the list of changes provided by the change detection module each change contains the following information:

```
<?xml version="1.0"?>

<POEM>
<TITLE>The Raven</TITLE>
<AUTHOR>Edgar Allan Poe</AUTHOR>
<DATE>1845</DATE>
<STANZA>
  <VERSE>Once upon a midnight dreary,while I pondered, weak and weary,</VERSE>
  <VERSE>Over many a quaint and curious volume of forgotten lore;</VERSE>
  <VERSE>While I nodded, nearly napping, suddenly there came a tapping,</VERSE>
  <VERSE>As of some one gently rapping, rapping at my chamber door.</VERSE>
  <VERSE>""Tis some visitor," I muttered, "tapping at my chamber door</VERSE>
  <VERSE>For Example this is deleted text</VERSE>
  <VERSE>Only this, and nothing more."</VERSE>
</STANZA>
<STANZA>
  <VERSE>Ah, distinctly I remember</VERSE>
  <VERSE>it was in the bleak April,</VERSE>
  <VERSE>And each separate dying ember wrought its ghost upon the floor.</VERSE>
  <VERSE>Eagerly I wished the morrow;vainly I had sought to borrow</VERSE>
  <VERSE>sorrow for the lost Lenore;</VERSE>
  <VERSE>For the rare and radiant maiden whom the angels name;</VERSE>
  <VERSE>Nameless here for evermore.</VERSE>
</STANZA>
</POEM>
```

Figure 5.7 Old version of an XML file involved in change detection.

- Content of the node that has changed
- Type of change whether it is an insert or a delete or a move
- The signature of the node that has changed
- The position of the node starting from the left of the tree
- The count of occurrences of the keyword in the old file in case of keywords
- The count of occurrences of the keyword in the new file in case of keywords

While traversing through the tree, we look for a match for the signature of and the position of the node that has changed. If the signature and position of a node encountered during traversal and that given in the list of changes are same we insert a tag for the type of change. The change detection provides us with the information if there is an

```
<?xml version="1.0"?>

<POEM>
<TITLE>The Raven</TITLE>
<AUTHOR>Edgar Allan Poe</AUTHOR>
<DATE>1845</DATE>
<STANZA>
  <VERSE>Once upon a time,while I pondered, weak and weary,</VERSE>
  <VERSE>Over many a quaint and curious volume of forgotten lore;</VERSE>
  <VERSE>While I nodded, nearly napping, suddenly there came a tapping,</VERSE>
  <VERSE>As of some one gently rapping, rapping at my chamber door.</VERSE>
  <VERSE>For Example this is the inserted text</VERSE>
  <VERSE>"'Tis some visitor," I muttered, "tapping at my chamber door</VERSE>
  <VERSE>Only this, and nothing more."</VERSE>
</STANZA>
<STANZA>
  <VERSE>Ah, distinctly I remember it was</VERSE>
  <VERSE>in the bleak December,</VERSE>
  <VERSE>And each separate dying ember wrought its ghost upon the floor.</VERSE>
  <VERSE>Eagerly I wished the morrow;vainly I had sought to borrow</VERSE>
  <VERSE>sorrow for the lost Lenore;</VERSE>
  <VERSE>For the rare and radiant maiden whom the angels name;</VERSE>
  <VERSE>Nameless here for evermore.</VERSE>
</STANZA>
</POEM>
```

Figure 5.8 New version of an XML file involved in change detection.

insert/delete/move in the node. The Dual Frame Presentation for the sample XML files shown in Figure 5.7 and Figure 5.8 is shown in Figure 5.9.

Assume the user is interested in the keywords “midnight” and ”December” in the sample versions of an XML page shown in Figure 5.7 and Figure 5.8. When the versions are given to change detection module, we derive that the keyword “midnight” is deleted from the old version and “December” is inserted in the new version. As discussed earlier we also get the signature and the position of the nodes that had the above mentioned changes in the XML tree. The signature of the node where “midnight” is deleted is “POEM-STANZA-VERSE” and the position is “4”. Similarly, in case of an insert of “December” the signature and position of the node containing the word is “POEM-

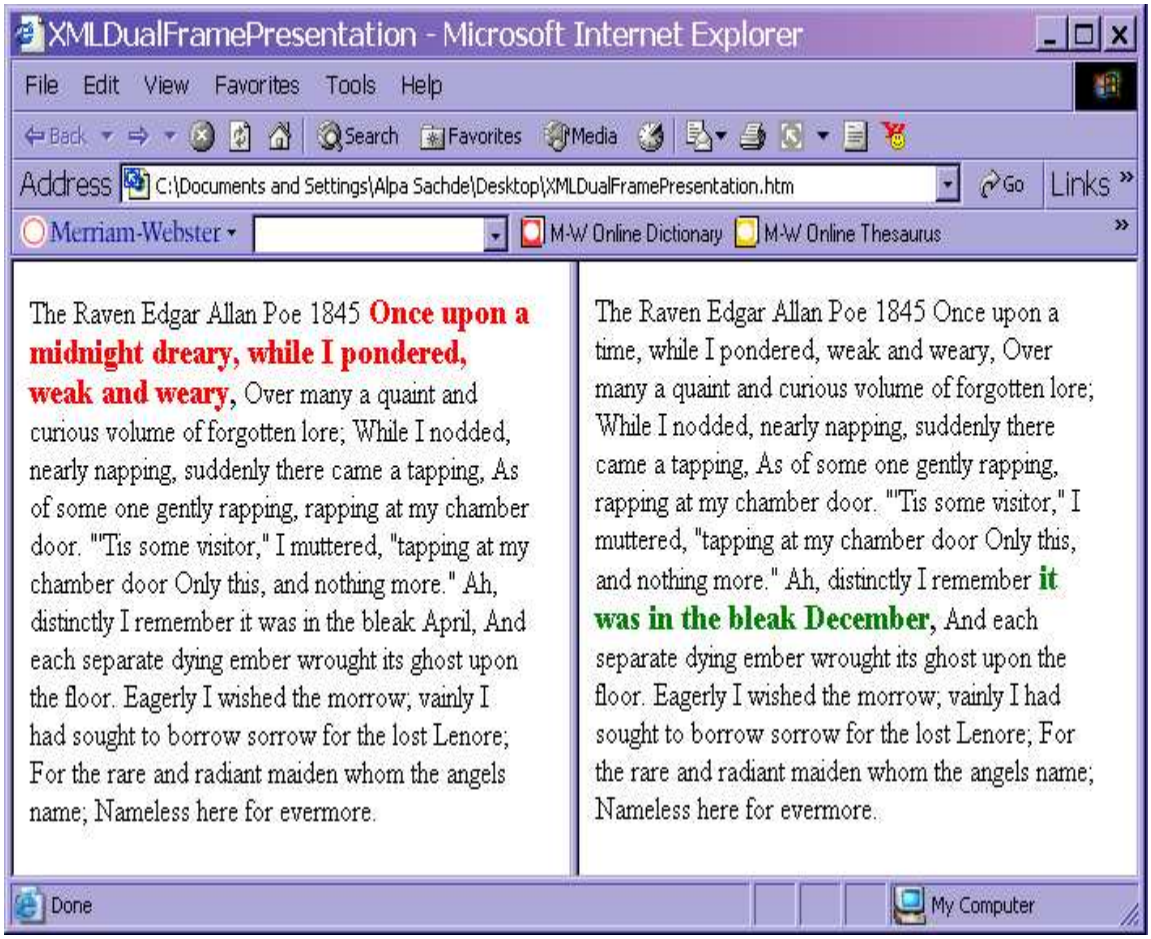


Figure 5.9 Dual Frame Presentation of changes in XML Pages.

STANZA-VERSE" and "12" respectively in the XML tree. Hence, while traversing the old XML tree we insert a node called "delete" as a child of the node that contains "midnight". Similarly, we insert a node called "insert" as a child of the node containing "December" in the new XML tree. The old and new XML trees with the "insert" and "delete" nodes are printed into temporary XML files. Both the temporary XML files are then given to the XSLT (XML Stylesheet Language Transformer) along with the generic XSL (XML Stylesheet Language) to display the XML with the highlighted nodes. Complete nodes containing the change are highlighted as finding the words in the node would be an additional overhead in case of large number of changes. Duplicate nodes

are also handled as we not only take into account the signature but also the position of the node in the XML tree involved in a change. In case of phrases the tree is realigned by the change detection algorithm, so the position and signature of the changed nodes with respect to the realigned tree is given as output. Hence it is not possible to traverse the tree formed by the original files and find the exact position and signature of change. Therefore currently the dual frame presentation handles "keywords" only.

5.4 Summary

This chapter presents presentation issues for HTML and XML pages. In this chapter, two schemes used for the presentation of changes detected by WebVigiL in HTML and XML pages are described in detail.

CHAPTER 6

IMPLEMENTATION

This chapter discusses various issues that affect the implementation of the modules addressed in this thesis. Section 6.1 deals with the implementation of the user interface. Section 6.2 explains the syntactic and semantic validation of the user specification and the flow from the user interface to Knowledge base to various modules of WebVigiL. Section 6.3 discusses the implementation details for the mechanism by which the types of notifications are handled. Finally a detailed discussion of the implementation of presentation of changes is given in section 6.5

6.1 User Interface And Dashboard

As WebVigiL [3] is envisioned for large network-centric environments such as internet, there is a need for a web-based interface for the users to place their monitoring requests.

6.1.1 User Interface

A web based user interface [17] as shown in Figure 6.1 is provided to the users to submit their profiles termed as sentinels. Sentinels are monitoring requests given by the users. The user interface is made simple and easy for navigation. The user interface is implemented using the client server technology. Server side scripting (JSP) is used to process the input given by the users. Every user that comes to WebVigiL web page can login into the system for placing a sentinel or retrieve information about his/her

previously-defined sentinels. Users are required to give certain details for a sentinel that follow the sentinel specifications as discussed in the section 3.1. Some of the details are:

- The page of interest to be monitored
- the type of content in the page to monitor.
- The user 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.
- The frequency with which the page has to be fetched can also be given. The options can be a fixed time interval where he/she probably expects a change or ask the system to fetch in which case the system learns and tunes the fetch frequency using a learning-based approach based on history.
- The medium of notification can be specified as email, PDA or fax.
- The frequency with which the user intends to receive notifications. It can be whenever the page changes or at certain specific time interval.
- The last detail is the previous relative version with which the user intends to compare the coming version of the page.

The users are given the options of changetype 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. Once the user submits the above-mentioned details to the system, the requests are then validated for syntactic and semantic correctness and the infix expression is converted to a postfix expression. The postfix expression that is generated after the conversion is ANYCHANGE:IMAGES:NOT:AND:LINKS:NOT:KEYWORDS:OR:AND. It is persisted and used by the WebVigiL system for change detection. A stack is used to convert the infix expression to a postfix expression.

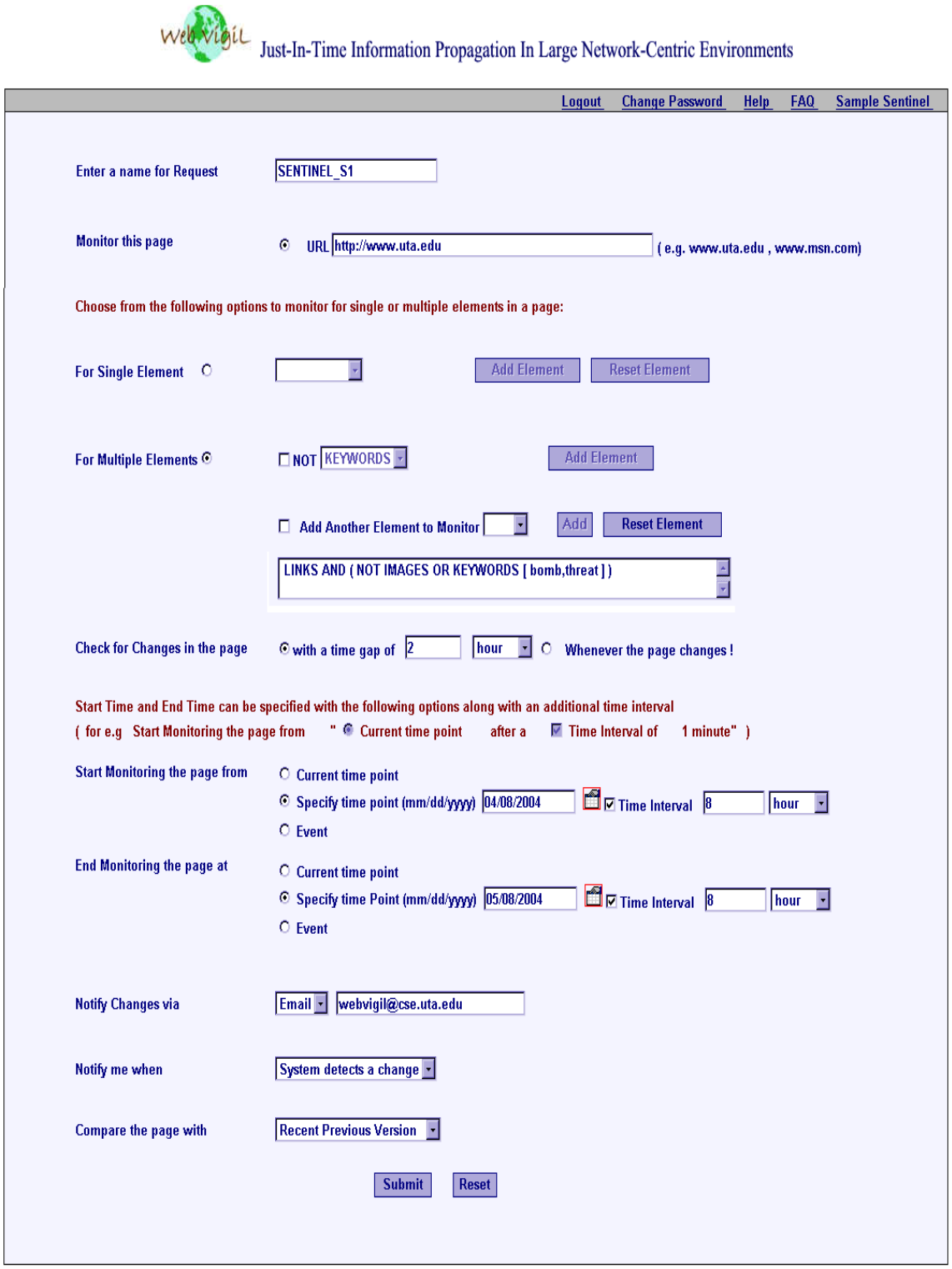


Figure 6.1 Web User Interface.

```
//Read each token (each operand, operator, or parenthesis) from left to right.
//If the next token is a changetype (operand) such as LINKS IMAGES,
ANYCHANGE, immediately append it to the postfix string.
//If the change type is KEYWORDS OR PHRASES, the next token is an
open square parathesis. Then until the closed square paranthesis is found the
keywords of interest are inserted into the a list.
//If the next token is an operator like AND, OR and NOT, pop and append to
the postfix string every operator on the stack until one of the following
conditions occurs:
        the stack is empty
        the top of the stack is a lef`t parenthesis (which stays on the stack)
        the operator on top of the stack has a lower precedence than the
current operator. The precedence of the operators is:AND and OR have the
first precedence and NOT follows them.
//Then push the current operator onto the stack.
//If the next token is a left parenthesis, push it onto the stack.
//If the next token is a right parenthesis, pop and append to the postfix string
all operators on the stack down to the most recently scanned left parenthesis.
//Then discard this pair of parentheses.
```

Figure 6.2 Infix to Postfix conversion of a composite change type expression.

```
public void sendEmail()
{
    SmtpClient client = new SmtpClient("mail.uta.edu");
    client.from(from);
    client.to(to);
    PrintStream message = client.startMessage();
    message.println("To: " + to);
    message.println("Subject:" + subject);
    message.println(msg);
    client.closeServer();
}
```

Figure 6.3 Part of the code to send mails to communicate with the users.

The pseudo-code for converting the infix expression to a postfix expression is as shown in Figure 6.2. A Java bean is used to store the values of variables for that session so that they can be retrieved very easily throughout the session. A SMTP mail client offered by Java is used to communicate with the client. The part of the code for sending the mail is shown in Figure 6.3. The table in Table 6.1 lists all the files used by user interface and a short description of the functionality of the programs in the files.

6.1.2 Dashboard

In order to use the full functionality of WebVigiL, it is important that the user be able to create and manage the sentinels as well as be able to retrieve selected changes as needed. In order to support all of the above functionality in a web-enabled manner, we have developed a "WebVigiL Dashboard" [17]. The basic idea is that the user can keep track of his/her sentinels, disable/enable them, delete them as well as use previously defined sentinels for creating new ones. Also, it should be possible to retrieve sentinels based on their attributes (show me all sentinels that expire on 04/12/2004, for example). In addition, one should be able to retrieve changes (both current and past) using the dashboard. The user can specify the number of changes he/she wants the system to persist so that the user can go to the dashboard and see when need be. The changes that the user feels are not of any use can also be deleted through the dashboard.

6.2 Validation

Syntax validation ensures the correctness of sentinel in terms of the defined grammar. In addition, the accepted sentinel should be semantically correct to strictly follow the well-defined semantics of the system. The validated policies are meaningful and used by different modules of WebVigiL. Once the request is validated, the details are populated

in a Knowledge Base, which is a database where all meta-data required by WebVigiL is persisted.

Syntactic validations are done at the client, thereby reducing excessive communication between the client and the server. For example, if the duration of a sentinel specified is incorrect, the sentinel will never start or will continue forever without ending. Some of the syntactic validations that are performed are:

- Users provide all the required attributes (default values are used for the rest if they are not input)
- If a date has been specified for a sentinel to start or end, it cannot be a date that has already passed
- Users provide a valid expression for change types if interested in a composite change
- Users give a valid email address for notification
- A positive integer is supplied for the frequency of a fixed interval fetch, notification and also for the relative version for comparison.
- Change Type: As WebVigiL supports composite changes (more than one type of change, the combination of the change types should be semantically meaningful.

Some of the semantic validations [5] where dependency on previous sentinels is involved are carried out at the server as it requires accessing and using the current state of another sentinel. For example a sentinel defined on Start/End time dependent on previous sentinels. As this information is with the server it has to be validated if the start/end time of the previous sentinel is a valid time with respect to current time. The validated sentinel is converted into an XML document and stored in the Knowledge-base for future reference. XML is becoming the standard format for exchanging data between diverse devices on a distributed environment in a platform and language neutral way. Hence it is selected for sentinel representation as WebVigiL is intended for large network-centric environments such as Internet.

6.3 Notification of changes

This section discusses in detail the implementation details of the notification of changes in WebVigiL. The three types of push-based notification frequencies that can be specified by the users, as discussed in section 3.2, are:

- Immediate
- Best-Effort
- Interval-Based

As discussed in chapter 3, changes are detected by the system and are inserted into the notify buffers which act as an interface between the change detection module and the notification module in the system. The two types of notify buffers maintained by the system where the detected changes are queued are *immediateNotifyBuffer* and *notifyBuffer*. Based on the choice of frequency of notification, the change detection module inserts the changes in the respective buffer in the form of notification objects. The Table 6.2 briefly describes the functionality of the above mentioned notify buffers.

Notification objects are inserted into the buffers by the change detection module which are retrieved by the notification module for further processing. The notification objects consists of the following properties:

- Sentinel Id
- List of changes
- Change history
- URL mapping
- Fetch type (best-effort or fixed interval)
- Notification Frequency (immediate, best-effort or fixed interval)

A notification thread waits on the buffers (and blocks) and serves both the buffers in the order of their priority as soon as an element arrives. The *notifyBuffer* is served only if *immediateNotifyBuffer* is empty. The notification objects are processed from the

Table 6.1 Files involved in user interface

File Name	Description
Login.jsp	Accepts the login information from a registered user
Loginvalidate.jsp	Validates the login information given by the users and navigates the users to the navigation page
Input.jsp	Form for the users to give the input requi red to place a sentinel and perform the syntactic validations for a valid input
Inputvalidate.jsp	Validates the input semantically and inserts the details of the sentinel into the KnowloedgeBase and sends the sentinel as an object to the WebVigiL server.
RetrieveSentinels.jsp	Manages and retrieves the sentinel details desired by the user.
RetrieveChanges.jsp	Retrieve and manage changes detected for a particular sentinel.
ChangePassword.jsp	Accepts the old and new password from the user
ProcChangePasswo rd.jsp	Processes the user request to change the password and updates the user information in the database
Forgotpwd.jsp	Accepts user id from the user to send the password by mail
ProcforgotPassword.jsp	Retrieves the password for the corresponding user id and sends it in email.
EmailClient.java	Used for sending mails to the users
PostFixConverter.java	Converts the infix expression for a composite change type into a postfix expression.

Table 6.2 Description of the Notification Buffers

Object	Description
immediateNotifyBuffer	Queues the changes which are to be notified immediately and fixed-interval notification
notifyBuffer	Queues the changes are associated with best -effort

```
public void run() {
   running=true;
   while(running) {
      NotificationObject notifObj=null;
         if(WebVigiLSystem.immediateNotifyBuffer.isEmpty()) {
            notifObj = (NotificationObject)WebVigiLSystem.notifyBuffer.get();
         }
         else {
             notifObj = (NotificationObject) WebVigiLSystem.immediateNotifyBuffer.get();
          }
          process(notifObj);
           notifObj = null;
   }
}
```

Figure 6.4 Buffer Selection Logic.

buffers in the order they arrived. As part of processing, a notification is sent if necessary (for e.g., immediate). The list of changes for each sentinel are then persisted in the change table in the Knowledge Base in the form of BLOBs. A BLOB consists of as many list of changes for a particular sentinel as mentioned by the change history which has been mentioned in chapter 4. The section of code that services the two above mentioned buffers is placed in the run method of the notification thread and is shown in Figure 6.4 When the users specify the option of fixed-interval notification, a Local Event Detector (LED) is used to serve interval-based notifications. Local Event Detector (LED) [18, 6, 7] is used for generating the start and end events. The methods of sentinel are used to raise the events to achieve the desired functionality. The Local event detector (LED), has been developed as a library that can be used to declare events and associate rules to be executed when events occur. Primitive events (as method executions) and temporal events (both absolute and relative time), as well as composite events (And, or, seq, and periodic used in WebVigiL) are supported in LED. Event-condition-action (or ECA) rules

Table 6.3 ECAAgent Class API

Method	Return Type	Description
CreatePrimitiveEvent	EventHandle	This method creates a primitive event of instance level.
createCompositeEvent	EventHandle	This method creates a composite event of instance level.
CreateRule	EventHandle	This method create s an instance level rule on the specified object instance
RaiseBeginEvent	static void	This method raises an event at the beginning of a method.

provide an elegant mechanism for supporting asynchronous notification based on events (temporal or otherwise).

ECAAgent class in LED contains the API to be used for generating events. The handle to ECAAgent is obtained as shown below:

```
import sentinel.led.*;
ECAAgent myAgent = ECAAgent.initializeECAAgent();
```

ECAAgent provides the API for the users to create class level and instance level primitive events, composite events and define rules on those events. Table 6.3 shows the methods used in creation of events and rules that is used for generating events.

The start and end events for a sentinel are generated by the ECA Rule generation module as soon as the sentinel is installed in the WebVigiL system. The start and end events are primitive events. The generation of events and rules for start and end of a sentinel are discussed in detail in [12] Apart from the start and end events, the ECA rule generator in WebVigiL generates events for fetch [15] and interval-based notifications. As explained in section 3.2.3, PERIODIC events are used to handle interval-based notifications. Periodic events are specified as (E1,t,E3), where E1 and E3 are the events that

function as an initiator and a terminator, and t is the time interval.The following API illustrates creation a PERIODIC (composite) event:

CreateCompositeEvent(EventType eventType,

java.lang.String eventName,

EventHandle leftEvent,

java.lang.String timeString,

EventHandle rightEvent)

Where "EventType" is a composite event which in this case is PERIODIC (LED) [18, 6, 7]. *leftEvent* is the initiator and *rightEvent* is the terminator which are primitive events. *timeString* is the interval at which the composite event is raised periodically. Once a periodic event is registered with LED and rules are associated with the corresponding events, the rules are executed by the LED when the events occur and are raised. The method for creation of a rule is shown below:

createRule(java.lang.String ruleName,

EventHandle eventHandle,

java.lang.String condition,

java.lang.String action)

Here condition and action define the method signature of a class that is to be called for the class instance, over which the periodic event is declared.

Figure 6.5 illustrates the creation of the periodic event and associated rule for an interval-based notification for a particular sentinel. The rule name provided here is an instance of a class called FINotification where the API for condition and action parts have been implemented. As we discussed earlier in section 4.3.3, the change table consists of a flag which is set when a change is inserted and reset when the notification for the last change has been sent. So the checkCondition API in the class checks if the flag is set to true which means that a change has been inserted after the last notification for that

```
public void generateNotificationPeriodicEventFixedTime(int id, String timeString)
{
        EventHandle[] start = ECAAgent.getEventHandles("StartEvent_" +
                              Integer.toString(id));
        EventHandle[] end  =  ECAAgent.getEventHandles("EndEvent_" +
                              Integer.toString(id));
        EventHandle notify  =  myAgent.createCompositeEvent(EventType.PERIODIC,
                              "NotifyEvent_"+Integer.toString(id),start[0],
                              timeString,end[0]);
        Rule rule =  myAgent.createRule(new FINotification(this.myAgent, timeString,id),
                        "NotifyRule_"+Integer.toString(id),notify,
                         "webvigil.ChangeNotification.FINotification.checkCondition",
                        "webvigil.ChangeNotification.FINotification.performAction");
 }
```

Figure 6.5 Generation of event and rule for interval-based notification.

sentinel. Hence the condition is evaluated to true if the flag is set. If the condition is evaluated to true the control passes to performAction API and a notification is sent for the last change detected for that sentinel and the flag is reset in the change table.

Table 6.4 gives a brief account of the functionality of the classes involved in notification. Various APIs are provided by the class "DBConnection" to insert and retrieve the BLOBs in the change table.

Table 6.4 Classes involved in notification

Class Name	Description
Notification	Generates the content to be sent in notification.
NotificationObject	Instance of this class is inset into the notification buffers.
FINotification	Instance of a rule fired when the notification event is invoked.
NotificationThread	A thread that constantly polls the notification buffers.

6.4 Version Management

A repository is implemented as part of the WebVigiL system to archive, manage and provide the versions of pages needed by the change detection and presentation modules. As the version manager is used by many modules, data structures of the version manager need to be synchronized when it is used and released when the module does not need it. The version manager offers the following functionalities:

- Provide APIs to the fetch module to check if a fetch is needed and sore the page
- Delete a page when no longer needed
- Provide mapping of URL to a directory structure

Interface with the fetch module: The fetch module interfaces with the version controller to decide if a fetch is needed when the fetch rule for a particular sentinel is fired. If there are two or more sentinels set on a same page with best-effort frequency, the page needs to be fetched once for all the sentinels. So the sentinels with best-effort fetch frequency are grouped for change detection [12]. The version manager maintains information about the version in a *Version* object. As search for version is going to be the most frequent operation by any module using the version manager, hash-tables have been chosen for storage for runtime information of a list of versions. Table 6.5 gives a brief account of the data-structures in version manger for storing runtime information of versions in the system.

The class *Version* has the following attributes:

- Version number
- Last Modified Date
- CheckSum
- Sentinel Id
- URL Mapping

Table 6.5 Data-structures involved in Version Management

Data Structure	Description
Version	Stores the information of a version of a particular page.
LatestHash	It stores the latest *Version Object* and has the URLMapping (URL) as its key
BestEffortHash	It stores the *Version Objects* of the versions used by the sentinels with best-effort fetch frequency and is hashed on URL
FixedIntervalHash	It stores the *Version Objects* of the version used by sentinels that are fetched with a fixed user-specified frequency. It has the sentinel Id as its key

The *LastestHash* is used by the fetch module to check for the existence of current version of a page for which the fetch rule is fired. It is hashed on URL for easy retrieval of the current version of that URL if it exists. *LatestHash* stores information about the current version of pages fetched for all the URLs in the system. The other two hash-tables used are *BestEffortHash* and *FixedIntervalHash*, which store a list of versions for a particular URL fetched by the system so far. *BestEffortHash* is hashed on URL as the sentinels for best-effort fetch for certain URL are grouped and same version will be used by all those sentinels. Hashing on URL saves a lot of redundant information stored which would have been the case if *BestEffortHash* was hashed on Sentinel Id. The frequency for best-effort fetch keeps changing as it is tuned by a learning algorithm based on the history of changes to that page [13]. The pages for sentinels with fixed fetch frequency are fetched at regular intervals and are individual to that sentinel hence the *FixedIntervalHash* hash sentinel id as its key.

Mapping of URL: Version manager also maintains two hash-tables for run-time information of the physical storage of versions in a directory structure. The class *Storage* consists of two hash-tables for maintaining the mapping and the reverse mapping of

the URL generated for storage. The class *StoreFileDirMapping* provides the APIs for mapping a URL to a directory structure and retrieving the URL given the mapping.

Deletion of Versions: The versions are stored in a linked list in such a way that the oldest version is at the end of the list and head of the list consists of the most recent version. The list is then stored in a data-structure called *VersionList*, which consists of the linked list of versions. Other details such as the change history, total number of versions in the list and deletion count as discussed in section 4.3.1 are also stored. A deletion trigger is calculated as shown in section 4.3.3 using the deletion count and change history in the respective hash-tables. If the total number of versions for a URL or sentinel in the list of version in either of the hash-tables reach the deletion trigger, the versions of that URL are checked for deletion. For deletion, the list for that URL in BestEffortHash and all the lists of the sentinels on that URL in FixedIntervalHash are checked using the deletion count and min version retrieved from the change table as explained in section 4.3.3. The main class of version manager is *VersionManager*. All the required data-structures are instantiated in this class and APIs provided are described in the Table 6.6

Thus the version manager module is implemented to archive the pages of interest. The deletion of pages have been implemented with an objective to strike a balance between the storage overhead and computation overhead. The module has been tested for its correctness using a test-bed of various inputs.

6.5 Presentation

The presentation module for HTML and XML has been implemented separately as the tags in HTML are used for presentation whereas the tags in XML define the content.

The change detection module detects changes and stores them in lists. The change is a vector containing three lists:

Table 6.6 APIs provided by the class VersionManager

Method	Return Type	Description
doesExist( )	Version /null	Checks the Latest hash for the latest version for a given URL
fillPage( )	Version	Differentiates between a best -effort request or fixed interval request to call store page
storePage( )	Version	Makes an entry for that version in the respective hash-tables according to the fetch type and calls writepage
writePage( )	---	Physically stores the page
checkFIBEHashforVersion ( )	Version	Checks the BestEffortHash or FixedIntervalHash for the given versio n information of a page
rephyChangeDetection ( )	Version	Provides the change detection with the version given the reference version and the relative offset for the old version of comparison to detect changes
getMapping( )	String	Calls the appropriate API of the class providing the mapping of a URL to a directory structure for efficient storage

- Insert list containing objects (links, images, words, phrases) that are inserted.
- Delete list containing the objects that are deleted
- move list containing objects that are moved.

A primitive change is associated with a single ChangeList whereas a composite (for e.g., LINKS AND IMAGES) is associated with multiple sets of ChangeList, that is, the collection of the ChangeList of all the constituent events. This collection is called ListOfChangeLists.

Change-only Approach for HTML and XML pages: This approach is the same for HTML and XML as it does not involve going through the contents of the files associated with the changes. The list of changes are inserted in the database in the form

```
if (changeType.startsWith("i")) {
        color = changeType.equals("i") ? "#66FFCC" : "#00FF00";
        buff += "<span style=\"background-color:" + color + "\">" +
                newFile.substring(occ_1,occ_1+word.length()) +"</span>";
}
else {
        color = changeType.equals("d") ? "#FFCCCC" : "#FF0000";
        buff  += "<span style=\"background-color:" + color + "\">" +
                newFile.substring(occ_1,occ_1+word.length()) +"</span>";
}
```

Figure 6.6 Dual Frame Logic.

of BLOBs. The changes are then retrieved from the lists mentioned above and placed in HTML tables. The presentation is generated on the fly when requested by the user.

Dual-Frame Approach for HTML pages: As the ChangeList contains information about the versions involved in the change detection, the information of the versions associated with the changes is derived from the corresponding ChangeList. The change types supported for presentation when a request is made for presenting the changes in a dual-frame presentation are “links”, “keywords” and “phrases”. The old version of the page involved in the change detection is traversed through to match for the changes in the delete list. Whenever a match is found the content that has changed is appended to an HTML tag that is used to highlight the changes associated with delete. Currently the background of the content is made “red” in case of a delete. Subsequently the new version of the file is traversed for the changes that are inserts. A different color is used to highlight the inserts. currently the inserts are highlighted making the background of the changed part of the content “green”. The part of the code that does the above is shown in Figure 6.6 In case of multiple occurrences of keywords, as HTML is unstructured and the changes are not position dependent, all the instances of the keywords in the content are highlighted. Thus the dual-frame approach for HTML pages is

implemented and tested on various input for changes in keywords, phrases and links. The class HTMLPresentation provides the APIs for making both the types of presentation of HTML pages mentioned above.

Dual-Frame Approach for XML pages: Changes detected for XML pages are also stored in ChangeList. The insert, delete and move lists in ChangeList are retrieved for the change that has to be presented. The associated version information is obtained as well. The old and new versions of the XML page are parsed using the Xerces Parser [19] and converted to a DOM [20] tree for traversing through it to locate the content that has changed. As the changes inserted in the lists are sorted in the order of their position in the tree (old or new), the tree is traversed only once to cover all the elements i the list. As the tree is traversed, for each node the position and signature in the tree is computed. The list is traversed to find a match until a position greater than the computed position is found. If a match is found a node for the type of change(insert/delete/move) is inserted between the leaf and the parent. Rather than traversing through the list of changes for each node, it is traversed only until the position of the node in the list is equal or greater then that computed while traversal. Once both (old and new) the trees are traversed and the changes have been made, an XSL is used to render them. A generic XSL (eXtensible Stylesheet Language) is made which recognizes the nodes that have been inserted with specific names(insert/delete/move). The XSL and the tree are given to the XSLT [16] (eXtensible Stylesheet Language Transformation) and an HTML file is generated where the nodes that have the change are highlighted. The look and feel of the rendered page is not guaranteed to be same as that of original pages as discussed in chapter 5. XMLPresentation class provides the API for invoking the presentation for both the *Dual Frame* and *Only Change* approaches. The class that is used for traversing the DOM [20] trees is XMLObjectExtractor which provides APIs for extracting the node

and inserting a new node between the parent and the leaf. In addition it also provides APIs for transforming the tree into an HTML page using the stylesheet.

6.6 Summary

In this chapter the implementation details of user interface and dashboard have been covered. Various details of notification of changes have been discussed. A detailed view of the archival and management of versions of pages of interest is given. In addition the implementation of presentation of changes has been discussed.

CHAPTER 7

RELATED WORK

7.1 Presentation And Notification

In this section we discuss some of the related techniques used for the presentation of HTML and XML pages.

Delta XML: It [21] developed by Mosnell provides a plug-in solution for detecting and displaying changes to contents between two versions of an XML document. They represent changes in a merged delta file [22] by adding additional attributes such as insert/delete to the original XML document.

Diff and Merge Tool: It [23] provided by IBM compares two XML files based on node identification. It represents the differences between the base and the modified XML files using a tree display of the combination in the left-hand, Merged View pane with symbols and colors to highlight the differences using the XPath syntax.

DOMMITT: It [24] is a UNIX diff utility tool that enables the users to view differences between the DOM representations of two XML documents. The Diff algorithms on these DOM [20] representations produces edit scripts, which are merged into the first document to produce an XML document in which the edit operations are represented as insert/delete tags for a user interactive display of differences.

AIDE: It presents a set of tools (collectively called AT&T Internet Difference Engine [25]) that detect when pages have been modified and present modifications to user through marked up HTML. AIDE uses HTMLdiff to graphically present differences using heuristics to determine additions and deletions between versions of a page.

AIDE uses the three most popular schemes in the dual frame approach, merged view and only change approach.

All of the techniques mentioned above produce edit scripts to apply on the documents to generate the changed latest version with changes. The requirement of the WebVigiL was to display the customized changes computed by the detection algorithms based on user input. The changes were at a finer granularity than a page. Hence the above techniques could not be used directly for resolving the issues for presentation in WebVigiL.

7.2 Version Management

This section gives a brief overview of the existing systems that deal with version management of documents. The two most popular schemes used for version management are RCS and SCCS. RCS [26] maintains the most recent version and uses edit scripts for retrieving the older version of the document. SCCS [27] is a time-based scheme where a unique version identifier is given to the object of interest and is retrieved based on that. The approach is somewhat similar to SCCS because the versions are given a unique version number based on meta-data.

Xyleme: It [28] uses a change-centric approach as opposed to data-centric approach which is more frequently adopted for database versioning. The representation based on deltas is used. Given two consecutive versions of a page, using a very efficient diff algorithm, the deltas are computed. Completed deltas that contain additional information are composed which can be reversed. Persistent identifiers are essential to represent and control changes. The authors believe that using deltas can be very useful to version results of continuous queries. In WebVigiL, the complete version of the page has to be stored to be given to the change detection module to compute

customized changes. Hence computing deltas to decide a fetch and storage will be an additional overhead.

WebCQ It [29] is a prototype system for large-scale web information monitoring and delivery, which makes use of the structure present in hypertext and the concept of continual queries. WebCQ is designed to discover and detect changes to the web pages and to provide a personalized notification of the changes to the users. Users' update to monitoring requests are modeled as continuous queries on the web. WebCQ change detection robot is responsible for discovering and detecting changes to web pages. WebCQ provides a trigger condition parameter by which the user can specify how frequently the change detection robot should be fired to check if any interesting changes have taken place and how soon the notification should be sent. WebCQ does not archive versions of documents, but it uses object cache update, wherein it stores the object in the old version that has changed in the new version. As WebVigiL provides the option of comparing the documents with not only the previous version but also a relatively older version.

AIDE A set of tools collectively are known as AIDE [25]. As a prototype, snapshot used a simplistic approach to authentication and version management. It is a CGI-based facility which could save versions of www pages in a central repository using the Revision Control System (RCS). RCS stores the most recent versions and uses reverse editing scripts. The state associated with each user was stored in a text file with one line per version checked in, indicating the URL and the modification time-stamp associated with the URL. To get a history of all versions stored for a URL, a CGI script would invoke the RCS system's *rlog* command and convert the output to HTML with the links to access each version and provide options to select the version of comparison. The mapping of version to the local file system is done by removing the "http:" prefix and converting characters in the remaining

string that might have special meaning to the UNIX file system into an innocuous % character. The computational overhead for processing the reverse scripts by the use of RCS will be significant if adopted in WebVigiL.

CVS It [30] is a software configuration management system which uses RCS [26], is a diff-based technique. It stores the most recent version of the document. It stores the last version together with backward deltas based on line-diff. These scripts describe how to go backward in the object's development. For any version, only current one is stored and extra processing is needed to apply the reverse editing scripts. In WebVigiL as older versions are required frequently by various modules, it will be a significant overhead to generate the older versions based on the reverse scripts.

Reference-Based Versioning: It [31] uses an approach based on time-stamps whereby an element appearing in multiple versions of the database is stored only once along with a compact description of versions in which it appears. It is tailored as an archiving tool for XML that is capable of providing meaningful and also support a variety of basic functions concerning the evolution of data such as retrieval of any specific version from the archive and querying temporal history of any element. The approach used in WebVigiL is a somewhat similar to this approach in terms of storing the complete version time-stamped rather than diff-based approaches.

The following are some of the other distinct characteristics of WebVigiL:

- Flexible specification of versions: All the above systems compute changes between two successive pages. In WebVigiL the user can explicitly specify the pages that can participate in change detection.
- None of the above systems except Xyleme [28] use the ECA (Event-Condition-Action) paradigm for monitoring the web and notifying the changes. ECA Rules help in adding new functionality to the system seamlessly.

- Properties of monitoring requests can be inherited: The user has the option of specifying the monitoring request to be dependent on the status of other monitoring requests. One can specify the start/end of a request to be the start/end of another request.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusion

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. WebVigiL is a system currently nearing completion at UT Arlington for providing an alternative paradigm for monitoring changes to the web (or any structured document).

The contributions of this thesis towards the system were in the design and development of the following:

- Version Manager which stores, manages and provides the versions of the pages for change detection.
- Presentation module that displays the detected changes for HTML and XML pages in an easy-to-understand manner.
- A web-based user interface for the user to manage and place sentinels(monitoring requests)
- A notification module that notifies the users of the detected changes with different frequencies.

The easy-to-use interface provided to the users has been developed and tested for correctness. The interface allows the users to place sentinels on the change type any-change, keywords, links, images and phrases. The users are authenticated to use the system. The users can specify a single change type or a combination of change types.

A facility to place a sentinel that starts or ends on previously defined sentinels has also been provided. The version manager has been developed and tested for interfacing with the fetch module to determine if a fetch is required. It also handles the mapping for the URL for the pages to be stored in a directory structure that they can be easily retrieved for change detection. As a complete content of the page is stored for each version of a page, in order to avoid storage overhead, the version manager checks for deletion periodically. It deletes the pages that are no longer required by the change detection module as well as the presentation module. In addition the system supports three frequencies at which notifications for the detected changes are pushed to the users. The IMMEDIATE notification is served as soon as the system detects a change for a particular sentinel. Similarly, the delivery semantics of BESTEFFORT are subordinate and subsequent to IMMEDIATE. Two separate buffers are used to serve the above notification schemes and the IMMEDIATE queue is prioritized over the BESTEFFORT queue. Local Event Detector is used for fixed-interval notifications. Events and rules are generated for periodic notifications with the interval provided by the uses. Finally the presentation module handles the presentation of changes in a manner that is easy for the users to decipher the changes. The two schemes used for HTML and XML are: *Change-only Approach* and *Dual-Frame Approach*. The manner in which the presentations for HTML and XML are made is totally different as the tags in HTML are used for presentation and the tags in XML define the content. *Change-only Approach* displays only the changes detected for a particular URL without the context. In *Dual-Frame Approach*, the changes are highlighted and displayed along with the rest of the content of the page in the old as well as new version of the page placed side-by-side in two frames.

8.2 Future Work

At present, a conservative approach for deletion of pages is used by the version manager. There are some versions that the systems maintains that may not be needed at any point of time. So the algorithm for deletion can be improved. Currently, notifications are served for a single sentinel at a time. The presentation of images is not handled currently as the images are not being fetched and stored by the system. The presentation of images will pose many problems that need to be investigated. The storage of images for every version of the page and then taking care of deletion will have a significant overhead on the system. detection of changes to pages (instead of pages) is being investigated currently.

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] American-Airlines, "http://www.aa.com."

[3] 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.

[4] N. Pandrangi *et al.*, "Webvigil: User-profile based change detection for html/xml documents," in *Proceedings 20th British National Conference on Data Bases*, Coventry, UK, 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] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] J. Jacob *et al.*, *WebVigiL: An approach to Just-In-Time Information Propagation In Large Network-Centric Environments.* Hawaii: Springer-Verlang, 2003.

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

[16] XSLT, "http://www.w3.org/tr/xslt."

[17] J. Jacob, A. Sachde, and S. Chakravarthy, "Cx-diff: A change detection algorithm for xml content and change presentation issues for webvigil," in *Proceedings of XSDM Workshop*, Chicago, October 2003, pp. 273–284.

[18] S. Chakravarthy *et al.*, "Hipac: A research project in active, time-constrained database management, final report," Xerox Advanced Information Technology, Cambridge, MA, Tech. Rep. XAIT-89-02, Aug 1989.

[19] Xerces-J, "http://xml.apache.org/xerces2-j/index.html."

[20] Document-Object-Model, "http://www.w3.org/dom/."

[21] R. Fontaine, "A delta format for xml: Identifying changes in xml files and representing the changes in xml," Delta XML, Europe, Tech. Rep., May 2001.

[22] R. Fontaine, "Merging xml files: A new approach providing intelligent merge of xml data sets," Delta XML, Europe, Tech. Rep., May 2001.

[23] XML-Diff-Merge-Tool, "http://www.alphaworks.ibm.com/tech/xmldiffmerge."

[24] DOMMIT, "http://www.dommitt.com."

[25] F. Douglis, T. Ball, Y. F. Chen, and E. Koutsofios, "The at&t internet difference engine: Tracking and viewing changes on the web," *World Wide Web Journal*, vol. 1, no. 1, pp. 27–44, January 1998.

[26] W. F. Tichy, "Rcs - a system for version control," pp. 637–654, July 1985.

[27] M. J. Rochkind, "The source code control system," in *IEEE Transactions on Software Engineering*, 1975, pp. 364–370.

[28] Xyleme, "http://xyleme.com/."

[29] 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.

[30] C.-V.-S. T. open standard for version control, "http://www.cvshome.org."

[31] S. Chien, V. Tsotras, and C.Zaniolo, "Efficient management of multiversion documents by object referencing," in *Proceedings 27th International Conference on Very Large Data Bases*, Roma, Italy, 2001.

[32] L. Li and S. Chakravarthy, "An agent-based approach to extending the native active capability of relational database systems," in *Proceedings, International Conference on Data Engineering*, Australia, 1999.

[33] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda, "Monitoring xml data on the web," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2001.

[34] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in xml documents," in *Proceedings, International Conference on Data Engineering*, San Jose, California, USA, 2002.

[35] S. Chakravarthy *et al.*, "Webvigil: Architecture and functionality of a web monitoring system," CSE Department, University of Texas, Arlington, TX 76019, Tech. Rep. CSE-2003-5, 2003.

[36] Mortis-Kern-System-Inc, "http://www.mks.com/products/wi."

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

[38] Changedetection, "http://changedetection.com/."

[39] Mind-it, "http://zen-eco.com/divingparadise/netminder.htm."

BIOGRAPHICAL STATEMENT

Alpa Sachde was born in Gujarat, India, in 1978. She received her B.S. degree from Sardar Patel University, India, in 1999. In the Spring of 2001, she started her graduate studies in Computer Science at The University of Texas, Arlington. She received her Master of Science in Computer Science from The University of Texas at Arlington, in May 2004.