WEBVIGIL: ADAPTIVE FETCHING AND USER-PROFILE BASED CHANGE DETECTION OF HTML PAGES

The members of the Committee approve the master's thesis of Official Naveen Pandrangi

Sharma Chakravarthy
Supervising Professor _____

Leonidas Fegaras _____

JungHwan Oh _____

WEBVIGIL: ADAPTIVE FETCHING AND USER -PROFILE BASED CHANGE DETECTION OF HTML PAGES

by

NAVEEN PANDRANGI

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

May2003

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his great guidance and support, and for giving me an opportunity to work on this project. I am also thankful to Dr. Leonidas Fegaras and Dr. JungHwan Oh for serving on my committee.

I would like to thank Anoop Sanka and Pratyush Mishra for maintaining a well-administered research environment and being so helpful at times of need. Thanks are due to Jyoti Jacob and Raman Adaikkalavan for their help and fruitful discussions during the implementation of this work. I would like to thank all my friends at ITLAB. I also thank my friends Sumanth Kodury, Manish Jodhani, Ramji Beera and Alpa Sachde for their support and encouragement.

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 am thankful to my parents and brother for their constant support and encouragement throughout my academic career without which I would not have reached this position.

March 25,2002

ABSTRACT

WEBVIGIL: ADAPTIVE FETCHING AND USER-PROFILE BASED CHANGE DETECTION OF HTML PAGES

Publication No. _____

Naveen Pandrangi, MS

The University of Texas at Arlington, 2003

Supervising Professor: Sharma Chakravarthy

Data on the web is constantly increasing. Many a times, users are interested in specific changes to the data on the web. Currently, in order to detect changes of interest, users have to poll the pages and check for the changes he/she is interested in. Efficient and effective change detection and notification is critical in many environments where a lot of resources are wasted in monitoring changes to the web manually. WebVigiL is a change monitoring system, which efficiently monitors changes to the page on behalf of the user and notifies the changes in a timely manner. It is a general-purpose, server based information monitoring and notification system.

This thesis focuses on detecting changes of interest to the users in HTML pages and the problem of fetching pages of interest by using the change pattern of the pages.

The CH-Diff algorithm is introduced for computing customized changes to an HTML page. An adaptive Best-Effort-Algorithm, based on history of observed changes, is introduced for fetching.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

The World Wide Web has become one of the most important media for sharing information resources and continuous to grow at an alarming rate. Users surfing the web, may either be searching for specific information or simply browsing the web. Different users may be interested in knowing changes to specific web pages (or even combinations there-of), and want to know when those changes take place. For example technical users would be like to monitor new technologies and new research results from engineering fields and business information of competitors. Such information would be essential for maintaining the competitive edge. Students may want to know when the web contents of the courses (they have registered for) change; users may want to know when news items are posted in a specific context (appearance of key words, phrases etc.) they are interested in. This will avoid periodic polling of the web (i.e., retrieval of one or more pages) to see whether the information has changed. Generally, to discover information of interest the users need to constantly monitor certain web sites and web pages.

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 are 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.

Today, information retrieval is mostly done using the pull paradigm, where the user is responsible for posing the appropriate query (or queries) to retrieve needed information. The burden of knowing changes to the contents of pages in interested web sites is on the user, rather than on the system. Although there are a number of applications (airlines, for example) that selectively send interested information periodically, the approach typically uses a mailing list to send the same information to *all* users. 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 still uses a naïve pull paradigm to refresh the screen periodically. In general, the ability to specify changes to arbitrary documents and get notified according to user-preferred ways will be useful for reducing/avoiding the wasteful navigation of web in this information age. In other words, users are interested in a variety of information from different sources and there is a real need for systems to be developed to support the task of automatically identifying changes and notifying the changes to the users in a timely and effective manner. The proposed system – termed WebVigiL provides a powerful way to disseminate information efficiently without sending unnecessary or irrelevant information. It also frees the user from having to constantly monitor for changes using the pull paradigm.

Active rules have been proposed as a paradigm to satisfy the needs of many database and other applications that require a timely response to situations. Event–Condition–Action (or ECA) rules are used to capture the active capability in a system. The utility and functionality of active capability (ECA rules) has been well established in the context of databases. In order for the active capability to be useful for a large class of advanced applications, it is necessary to go beyond what has been proposed/developed in the context of databases. Specifically, extensions beyond the current state of the art in active capability are needed along several dimensions:

1. Make the active capability available for non-database applications, in addition to database applications;

2. Make the active capability available in distributed environments

3. Make the active capability available for heterogeneous sources of events (whether they are databases are not).

We believe that some of the techniques developed for active databases, when extended appropriately along with new research extensions will provide a solution to the above class of problems. In addition, there is the theoretical foundation for event specification, and its detection in centralized and distributed environments. The main objective here is to have a selective propagation approach that can be applied to web and other large-scale network-centric environments using the existing active technology. In chapter 2 we will discuss the existing change detection and notification systems focusing on the areas where WebVigiL differs from the rest. In chapter 3 we will present a detailed overview of the current architecture. The main contribution of this thesis is presented in chapters 4 through 7 where we address the problem of detecting changes to HTML pages and various mechanisms for fetching of pages. We present a history-based algorithm for tuning fetch interval to change interval of a page and evaluate it experimentally.

CHAPTER 2

RELATED WORK

In this chapter we discuss currently used various change detection mechanisms for detecting changes to HTML pages. We also discuss the approaches taken by various web-monitoring systems for fetching pages of interest to the users.

2.1 Detecting Changes to HTML Pages

Change management is done by many commercial editing packages Microsoft Word , for example, has a “revisions” feature that can capture and keep track of simple updates, inserts, and deletes of text. Similarly, WordPerfect has a “mark changes” facility that can detect changes based on how documents can be compared (on either a word, phrase, sentence, or paragraph basis). Other previous work in change detection has dealt only with flat-files [1] and in [2, 3] authors detect changes between strings using the longest common subsequence [4] algorithm and consider only insertion and deletion operations. In this chapter we will discuss existing systems that, address detecting changes to HTML pages.

2.1.1 HTMLdiff

AIDE[5] presents a set of tools (collectively called AT&T Internet Difference Engine) 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. Here the authors view a HTML document as a sequence of sentences and sentence-breaking markups. A

sentence is a non-recursive set of words and non-sentence-breaking markups. Sentence breaking markups separate sentences from each other and from collections of sentences. All markups are represented and are compared. A token is either a sentence-breaking markup or a sentence, which can be a sequence or words and non-sentence-breaking markups. HTMLdiff uses weighted LCS algorithm [4] to compare two tokens and computes a non-negative weight reflecting the degree to which they match. A weight equal to 1 is taken as a match. HTMLdiff uses token length (excluding markups such as <B>, <I>) as a comparison metric along with computation of LCS of the two sentences. This approach may be expensive computationally as each sentence may need to be compared with all sentences in the document. Thus in situations where the user is interested in change to a particular phrase, HTMLdiff will end up computing changes to the whole page, resulting in excessive computational cost.

2.1.2 WebBeholder

WebBeholder[6], a community of service provider agent and a number of mobile agents representing users; aims at tracking and viewing changes to the web. It is a cooperative agent community framework that provides open services on finding and displaying changes on the web. The agents in the community interact with one another to achieve change detection and presentation on the web. The service provider agent is responsible for retrieving and comparing HTML documents. A Difference Engine is used to compare and summarize change information. The authors present an algorithm called Longest Common Tag Sequence (LOGTAGS) for finding the right places for context comparison within a pair of HTML document, to determine meaningful changes. The authors view a HTML document as a string of markup tags and context. The context and tag sequence is treated separately while keeping the sequence right. The authors claim the

algorithm can compare the context to its pair at the right positions because the sequence of the markup tags are checked and recognized. Here the granularity to which a change can be detected is limited to a page level.

2.1.3 WebCiao

WebCiao[7] a website visualization and tacking system focuses on the structure changes on a website. Change in the links of a page constitutes a structural change. WebCiao tracks the structures of websites by creating an archived website database and differencing a page with the database. It uses a database differencing tool called diffdb to detect structural changes and creates a difference database that consists of all pages and links classified as added, deleted and changed. We intend to address change at both textual level and structure level in WebVigiL.

2.1.4 Clone Analysis

In [8] the authors propose an approach for detecting similar pages using the concept of clone analysis. Considerable work [9, 10] has been done investigating methods and techniques for detecting duplicated portions of code or portions of similar code in procedural software systems. The authors suggest websites to be good candidates for clone proliferation, and propose an approach for clone analysis for websites based on Levenstein distance [11]. Levenstein distance represents the minimum number of insert or delete operations required to transform a first string into a second string; its value expresses the degree of similarity of the two strings. But the computation of Levenstein distance for websites requires that an alphabet of distinct symbols (HTML tags) in pages be defined. In HTML, the same sequence of attributes can refer to different tags, there by producing false positives if not properly linked to correct tags. The authors resolve this problem by refining the preliminary alphabet

(including all HTML tags) and by substituting each tag in the alphabet with an equivalent tag. Elimination of data formatting tags does further filtering of the alphabet (A*). In the approach suggested by the authors for detection of clones, the HTML files are parsed, tags are extracted from them and composite tags are substituted with their equivalent tags. The resulting strings are processed to belong to the alphabet A*. A distance matrix is then computed for these strings based on Levenstein distance. This method is computationally quite expensive as computation of Levenstein distance involves evaluation of all possible alignments between strings before an optimal alignment can be determined. Moreover, it is not always possible to have a predefined alphabet for all the tags in HTML pages given the nature of their development.

The above mentioned change detection approaches for HTML pages do not support customized changes based on user interest and are computationally expensive.

2.2 Fetching Pages of Interest for Change Detection

Every Web monitoring tool needs to fetch the page based on the user set fetching frequency; alternatively it has to tune its polling frequency to the change frequency of the page. The problem of estimating change frequency has been long studied in statistics community. Most of the previous work assumes complete knowledge of change history, which is not true for most of our intended applications/usage. In data warehousing context, a lot of work has been done to efficiently maintain materialized views. The materialized views are usually updated during off-peak hours, to minimize the impact on the underlying source data. As the data size grows, it becomes difficult to update the view within the limited time-window. It is therefore required to estimate how often an individual data item changes, so that we may selectively update only the items that are likely to have changed. However most of the work in data warehousing is focused on issues such as minimizing the size of the view

while reducing the query response time. The problem of knowing changes to web pages is very different from the data warehouse problem and hence those techniques are not applicable here. Several tools are available to assist users in tracking changes to web pages. Most of these tools are domain based and offer tracking service on a client's machine. Netmind and TracerLock are server-based tools. Some tools offer tracking service on a specific or a set of pages instead of on any(arbitrary) registered page (Amazon's new book alert).

In [12], the authors suggest that many online data sources are updated autonomously and independently and thus make the case for estimating the change frequency of the data, to improve web crawlers and web caches. A web crawler is a program that retrieves web pages, commonly used by search engines or web cache or web monitoring systems. Various scenarios are presented, where different applications have different requirements on the accuracy of the estimated change frequency. The authors assume that an element (which can be a web page or particular portion in page) changes according to a Poisson process. A Poisson process is often used to model a sequence of random events that happen independently with a given rate over time. The authors experimentally deduce that a given set of pages follow the Poisson process and thus assume that the set of pages change by a random process on average. Thus based on the description of a Poisson process, a notion of frequency estimator based on the number of occurrences of changes in a given interval (average frequency or rate with which a change occurs) is introduced. Many more new estimator calculation methods were proposed that compute change frequency reasonably well with incomplete history. Though the approaches proposed result in determination of reasonable frequency estimators, these are not adaptable, i.e., even if initially a certain frequency is chosen, we may want to adjust it on the fly, when the estimated change

frequency is different from the initial guess. Issues of how and when exactly should the frequency be adjusted is not addressed in [12].

2.2.1 WebMon

WebMon monitoring system [13] is proposed for tracking web information over Internet. In WebMon the user can specify the web page to be monitored, select the monitoring function and state the monitoring frequency to be used. When the monitored page is updated or changed according to the specified criteria, the updated results are automatically stored into the user's personal folders and notifications are sent out to the users informing about the updates. The update checking process of the system is responsible for retrieving the page from the web. The user can specify monitoring frequency such as daily, weekly, etc. The system is activated based on the monitoring frequency specified by the user to check the updates of the web pages being monitored. Here the lower granularity in specifying the frequency is limited to one day. Hence scenarios where the user wishes to track a page at a granularity of hours, minutes cannot be handled by the system. In [13] the authors state that an immediate checking facility is provide by the monitoring system to enable the users to override the predetermined frequencies of checking in order to carry out an immediate detection of changes to the pages being monitored. But the issues surrounding such adaptive retrieval of pages based on the change frequency are not addressed.

2.2.2 WebGUIDE

WebGUIDE[14] is a system for exploring changes to web pages and their structure; users can explore the differences between pages with respect to two dates. Differences between the pages are computed automatically and summarized in a new HTML page and

the differences in link structure is shown via a graphical representation. WebGUIDE uses a centralized repository service to archive versions of pages. The repository is responsible for retrieving the pages that the users explicitly request. Further more, each page is retrieved based on two parameters ”last check” and “frequency of checks”. The parameter “last check” of a page (the time when the last modification date was obtained) is used to determine when the page should next be checked; “ frequency of checks” defines the minimum frequency at which a page is retrieved i.e., different users may request different minimum frequencies to check a URL. A minimum across all the users is taken as the frequency of fetching for that page. The draw back with these parameters is that even if a user is interested with changes in the specified interval (relevant to the user) the system will end up presenting him changes based on the minimum fetch frequency determined by the system, there-by leading to presentation of changes that are not of importance to the user.

2.2.3 Xyleme

In [15] the authors present a Dynamic Warehouse for Web: Xyleme, monitors the flow of incoming documents. The flow of documents consists of XML pages and HTML pages (monitors the flow of documents that are discovered and refreshed). The authors present a subscription language for specifying the pages to be monitored. Depending on type of information requested by the user the pages are monitored using either monitoring or continuous query. For query of type “monitoring”, changes to a page are discovered when the system reads the page and for “continuous queries” changes are be discovered by regularly executing the same query and discovering the changes.

2.2.4 WebCQ

WebCQ[16] 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 monitoring requests are modeled as continual 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 happened. When triggered the robot uses a static page crawler to fetch static pages and a dynamic crawler for handling the remote fetching of dynamic pages. But in cases where the user is not aware of the frequency of change to a page, the system provides no addition capability such as fetching the page for the user pro-actively by estimating the change frequency to the page.

CHAPTER 3

CURRENT ARCHITECTURE

WebVigiL is a change detection and notification system, that can monitor and detect changes to unstructured documents in general. The current work addresses HTML/XML documents that are part of a web repository. WebVigiL aims at investigating the specification, management, and propagation of changes as requested by the user in a timely manner while meeting the quality of service requirements. Figure 3.1 summarizes the high level architecture of WebVigiL. Users specify their interest in the form of a *Sentinel* that is used for change detection and presentation. Information from the sentinel is extracted and stored in a data/knowledge base and is used by the other modules in the system.

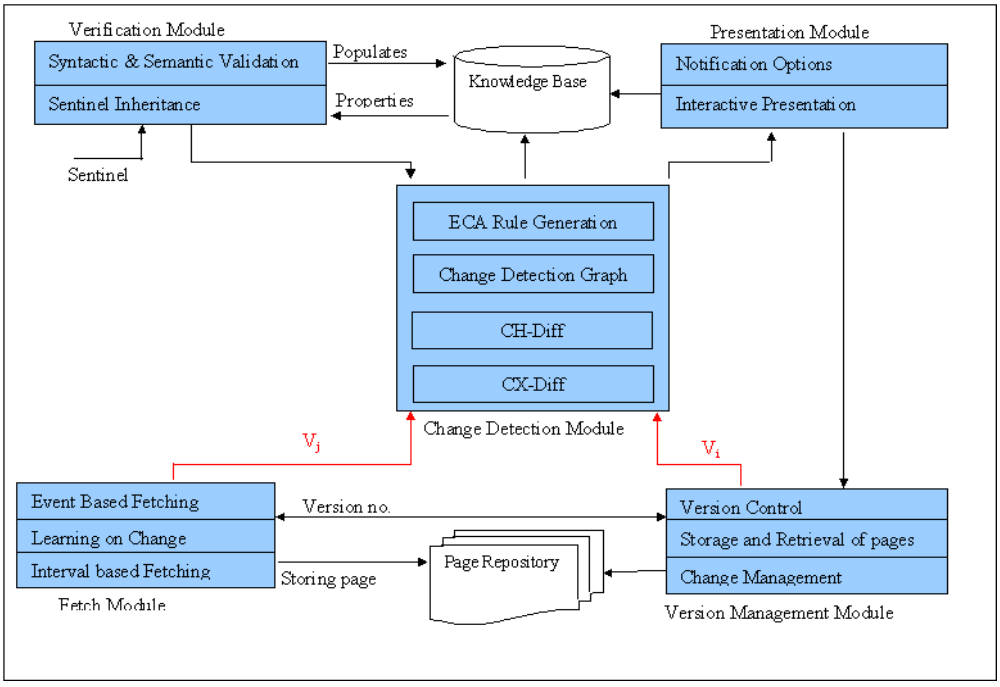


Figure 3.1 WebVigiL Architecture

The functionality of each module in the architecture is described briefly in the following sections.

3.1 Sentinel

WebVigiL provides an expressive language with well-defined semantics for specifying the monitoring requirements of a user, pertaining to the Web. Each monitoring request is termed a *Sentinel*. The specification language supports the following features:

- A suite of change types at appropriate levels of granularity that are of interest to a large class of users. For example, changes only at the level of a page may be overkill in many cases. One may be looking for changes to keywords or phrases of interest.
- Ability to monitor a page based on the actual change frequency, or at a user-specified frequency. The specification of the actual change frequency relieves the user of knowing when the page changes and requests the system to do its best effort.
- Ability to specify detection of multiple types of changes on a page.
- Notification frequency either as best effort or with pre-determined frequency.
- Multiple ways to compare changes (e.g., pairwise, every n, or moving n).
- Specification of a sentinel in terms of previously defined sentinels. Also, start and stopping of a sentinel may be based on other sentinels. This provides a mechanism for tracking correlated changes.

For example consider the Scenario: Jill wants to be notified daily by e-mail about changes to links and images to the page “http:// w ww. cnn. com” starting from December 2, 2002 to January 2, 2003. The sentinel generated for the above scenario is as follows:

Create Sentinel s1 Using http:// w ww. cnn. com

Monitor all links AND all images

Fetch 2 day

From 12/02/02 To 01/02/03

Notify By email jill@aol.com Every 4 day Compare pairwise

3.2 Verification Module

Verification module provides the required communication interface between the system and the user for specification of sentinels. User requests (sentinels) are processed for syntactic and semantic correctness. Valid sentinels are populated in Knowledge base (Oracle is used currently) and a notification of the valid sentinels is sent to change detection module. In general the functionality of verification module can be summarized as

- Load balancing of syntactic validation between client and server, thereby reducing excessive communication like validating start date set to a date in past at the client's end than checking at the server.
- Semantic validation of sentinels at the server, as the dependency information specified in the sentinel is available at the server. For example if the start of a sentinel s1 was specified on the end of another sentinel s2, and at the time of specification if s2 had already expired an error should be thrown to the user.

3.3 Knowledge Base:

Knowledge Base is a persistent repository containing meta-data about each user, number and names of sentinels set by each user, and details of the contents of the sentinel

(frequency of notification, change type etc.). The details of a sentinel need to be stored (in a persistent and recoverable manner) as several modules use this information at run time. For example, 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.

To satisfy all the above requirements, the metadata (the WebVigiL Knowledge Base) generated and used by different modules is stored in a relational DBMS. The monitoring request is parsed and sentinel properties are extracted, validated and stored in the KB. For example, the following parameters are stored for notification: the frequency of notification and the mechanism to notify the user. In addition, important run time parameters computed by different modules, such as the status of the created sentinels and parameters of the change detection module are also persisted in the KB. Finally, relational database provides mechanisms to extract the required information in a convenient manner in the form of queries or using the JDBC Bridge.

3.4 Change Detection Module

Every valid user request arriving at WebVigiL, initiates a series of operations that occur at different points in time. Some of these 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 ECA rules [17, 18] to perform some of these operations. This module is responsible for:

1. Activating and deactivating sentinels
2. Maintaining Change Detection Graph
3. Generating Fetch rules.

3.4.1 Activation/Deactivation

During its lifespan, a sentinel is active and participates in change detection. A sentinel can be disabled (does not detect changes during that period) or enabled (detects changes). By default, a sentinel is enabled during its lifespan. The user can also explicitly change the states of the sentinel during its lifespan. The start/end of a sentinel can be time points or events. When a sentinel's start time is *now*, it is enabled immediately. But in cases where the start is at a later time point or depends on another event that has not occurred, we need to enable the sentinel only when the start time is reached or the event of interest has occurred.

3.4.2 Change Detection Graph

When a page is fetched, for every sentinel that is interested in that page, change is computed and notified to the user. In situations where there are two or more sentinels interested in the same type of change on the same page we have to compute the change more than once. We avoid this by capturing the relationship between the pages and sentinels, and grouping the sentinels on their change and page. Hence all sentinels interested on the same type of change and on the same page are grouped together. In order to represent this relationship we construct a change detection graph. The change detection graph for the sentinels s1 and s3 is shown in Figure 3.2. The different types of nodes in the graph are as follows:

- URL node: A URL node is a leaf node that denotes the *page of interest*. The number of URL nodes in the graph is equal to the number of *distinct* pages the system is monitoring at any particular instant of time.
- Change type node: All level-1 nodes in the graph belong to this category. This node represents the *type of change* on a page (all words, links, images, keywords, phrases, table, list, regular expression, any change).
- Composite Node: A Composite node represents a combination of change types. All higher-level nodes (> level-1) in the graph belong to this type. Currently, we support composite changes on a single page. We plan on extending this to multiple pages.

In the graph, to facilitate the detection and propagation of changes, the relationship between nodes at different levels is captured using the subscription/notification mechanism. The higher-level nodes subscribe to the lower level nodes in the graph. This subscription information is maintained in the subscriber list at each node. At the URL node, this list contains the references to the change type nodes. At the change type nodes each sentinel will have a subscriber that will contain the references to the composite nodes. When a page is fetched, the associated URL node is notified about the page. The URL node propagates this page to all the change type nodes that have subscribed to it. Finally, at the change type nodes the change is computed between the current page received and an appropriate reference page (based on the compare option) that is fetched from the page repository. If there is any change then the sentinels subscribed to it are notified. When this change type is a part of a composite change, those composite nodes are notified.

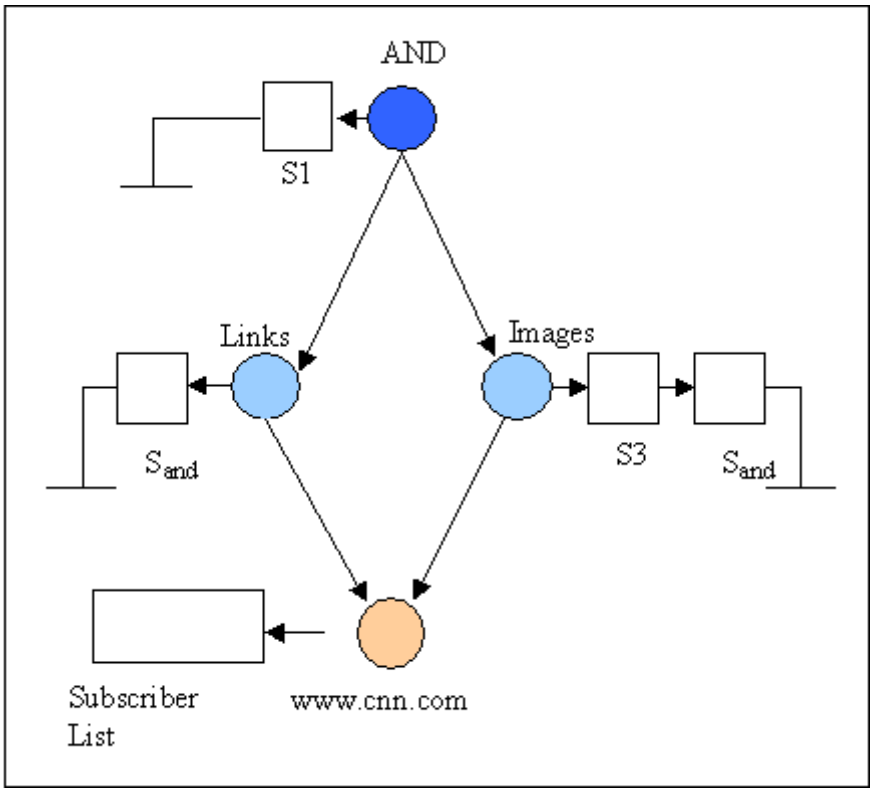


Figure 3.2 Change Detection Graph

3.4.3 Detection algorithms

A detection algorithm associated with each change type node computes changes between two versions of a page with respect to that change type. For a change to be detected, the object of interest is extracted from the given versions of the page depending upon the change type. Change detection algorithms have been developed to detect different types of changes to HTML and XML pages. The change types currently supported are: links, images, all words, keywords, phrase and regular expressions. Change to links, images, words and

keyword(s) is captured in terms of insertion or deletion. For phrases in addition to insertion/deletion updates are also detected.

3.5 Fetch Module

The Fetch Module of WebVigiL is responsible for retrieving the pages registered with it and thus serves as a local wrapper for the task of fetching pages depending upon the user set fetching policy i.e., fetching a page after a specified interval (set by the user) or fetching the page on change (the system determines the frequency of fetching based on change frequency of the pages). The Fetch module informs the version controller of every version it fetches, stores it in the page repository and notifies the change-detection-graph of a successful fetch. The wrapper fetches the page only when there is change in the properties of the pages. By properties, we mean the size of the page and last modified time stamp. When there is a change in time stamp of the page with an increase or decrease in page size, the wrapper fetches and caches the page. In cases where time stamp is modified, but the page size remains the same, the wrapper fetches and calculates the checksum of the page. This version of the page is cached only if the calculated checksum differs from the checksum of the cached (previous) version of this page.

3.6 Version Management

An important feature of WebVigiL architecture is its centralized server-based repository service (Version controller) that archives and manages versions of pages. WebVigiL retrieves and stores only those pages needed by a sentinel. The primary purpose of the repository service is to reduce the number of network connections to the remote web server, there by reducing network traffic. When a remote page-fetch is initiated, the

repository service checks for the existence of the remote page in its cache and if present, the latest version of the page in the cache is returned. In cases of cache miss, the repository service requests that the page be fetched from the appropriate remote server. Subsequent requests for the web page can access the page from the cache instead of repeatedly invoking a fetch procedure.

The repository service reduces network traffic and latency for obtaining the web page because WebVigiL can obtain the “Target Web Pages” from the cache instead of having to request the page directly from the remote server. The quality of service for the repository service includes managing multiple versions of pages with out excessive storage overhead.

3.7 Presentation Module

The principal functionality of this module is to present clearly the detected differences between two web pages to the user. Therefore, computing and displaying the detected differences is very important.

3.7.1 Change Presentation

Different methods of displaying changes used by the existing tools are: i.) Merging two documents, ii.) Displaying only the changes iii.) Highlighting the differences in both the pages. Summarizing the common and changed data into a single merged document has the advantage of displaying the common portions only once. The disadvantage of this approach is that it is difficult for the user to view the changes when they are large in number. Displaying only the computed differences is a better option when the user is interested in tracking changes to multiple pages or when the number of changes is large. But, highlighting the differences by displaying both the pages side-by-side is preferable for changes like “any

change" and "phrase change". In this case, the detected differences can be perceived better if the change in the new page is shown relative to the old page.

Because WebVigiL will track multiple types of changes on a web page, and eventually notify using different media (email, PDA, laptop etc.), combination of all presentation styles discussed above will be relevant, as the information to be notified will vary depending on factors like notification method, number of detected differences and type of changes.

3.7.2 Change Notification

Users need to be notified of detected changes. The mechanism selected for notification is important especially when multiple types of devices with varying capabilities are involved. What, when and how to notify are three important issues for notification.

3.7.2.1 Presentation Content

Presentation content should be concise and lucid. Users should be able to clearly perceive the computed differences in the context of his/her predefined specification. The notification report could contain the following basic information:

- The change detected in the latest page relative to the reference page
- User specified type of change like "any change", "all words" etc.
- URL for which the change detection module is invoked.
- Small summary explaining the detected changes.

This could include statuses of changes such as insert, delete and changed for certain type of user-defined types of changes like "images", "all links" and "keywords" or/and the different timestamps indicating the modification, polling, change detection and notification

date. The size of the notification report will depend upon the maximum information that can be sent to a user by satisfying the network quality of service requirements.

3.7.2.2 Notification frequency

A detected change can be notified in two ways: i) Notify immediately when the change is detected ii) Notify after a fixed time interval. The user may want to be notified immediately of changes on particular pages. In such cases, immediate notification should be sent to the user. 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. The notification has to be sent to the user taking into consideration the QoS constraints. The system should incorporate the flexibility to allow users to specify the desired frequency of notification. For example, in sentinel s1, Jill wants to be notified every 4 days, irrespective of when the changes are detected.

3.7.2.3 Notification methods

Different notify options like email, fax, PDA and interactive way, can be used for notification. Interactive is a retrieval-based notification approach where the user retrieves the detected changes as and when needed. A dashboard will be provided to the user to view and query the changes generated by his/her sentinels.

CHAPTER 4

EVENT-BASED FETCHING

With the explosive growth of the Internet, many data sources are available online. Most of the data sources are autonomous and are updated independently of the clients that access the sources. For example, business users would like to know changes to relevant information like new products released by competitors, promotional campaigns, and casual users would like to monitor the release of new songs movies etc. Since the sources are updated autonomously, the clients usually do not know exactly when and how often the sources change. In this chapter we will address this problem and discuss in detail the approach taken for estimating the change frequency of the pages of interest to users subscribed with WebVigiL. WebVigiL monitors only those pages that are registered with it. For this purpose, it has to fetch the pages when a change in metadata (such as the last modified time stamp or checksum) is detected. In this chapter we will discuss the criteria for fetching, the paradigm used for fetching, and present an adaptive algorithm to tune the fetching interval to change interval and experimentally analyze its performance against the ideal and naïve approaches. Figure 4.1 shows the various components of the fetch module responsible for fetching the pages. Depending upon the user-set fetching policy i.e., fetching a page after a specified interval (set by the user) or fetching the page on change (the system determines the frequency of fetching based on change frequency of the pages). The Fetch module informs the version controller of every version it fetches, stores it in the page repository and notifies the change-detection-module of a successful fetch. In this chapter we present our approach to estimate the change frequency of a page.

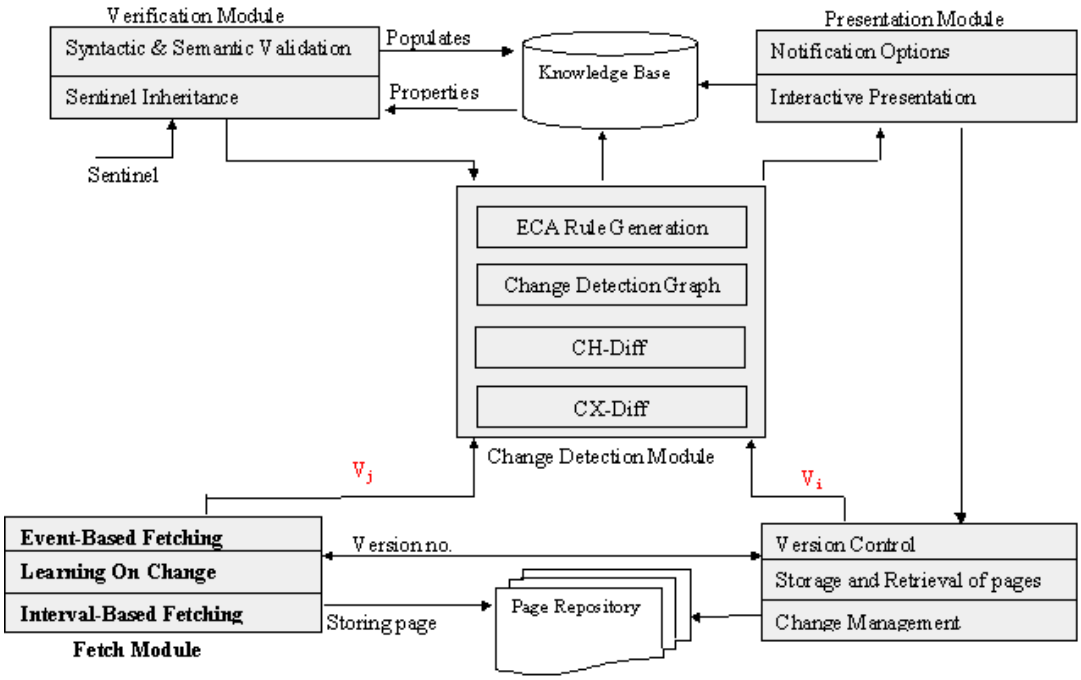


Figure 4.1 System Architecture

Further more, we assume that from repeated access of the page we can estimate the actual changes to the page and estimate its change frequency. However, there exist many challenges in estimating change frequency of a page including the following:

1. Estimating the change frequency based on the observed history of change.
2. Relevance of observed history to the actual change history of the page.
3. Availability of page properties, which aid in detecting changes.
4. Nature of change to the page (content vs. context).

Both content and context based changes to a page are taken as a change to the page and the page is fetched. In addition, we fetch a page (HTML/XML) that is available under default security mode (pages under secure login are not considered in this thesis).

4.1 When to Fetch

The interval at which we fetch a page determines the number of times a page is retrieved for change detection. Larger intervals will miss changes and smaller intervals will result in excessive network connections and communication cost in addition to the detection of changes. Ideally the fetching interval should be synchronized with the change frequency of the page. Following are two approaches that can be used to address the above problem.

- **Naïve Method**: Since the pattern of page change (intervals at which the page is changing) is not known, one approach could be to poll the page at a pre-defined interval. T When the actual change interval is greater than the fetch interval, this approach will not miss any changes, provided the change interval is greater than the fetch interval. But for those pages that are changing infrequently (i.e., at intervals far greater than the fetch interval) this approach will result in excessive network connections and data transfers, as the nature of change is not taken into consideration in determining the fetch interval. On the other hand, when the actual change interval is less than the fetch interval, some changes will not be detected.
- **Adaptive Method**: The basic idea behind this approach is to adapt the fetching frequency to the actual change frequency based on what has been observed. In this approach, we take the history of change patterns on a page, as observed and recorded in the previous fetch cycles to determine the interval after which the page should be fetched. Initially we start with a predefined frequency as in the Naïve method and try to converge on the actual change interval.

Since it is not likely that a page is changing at a constant frequency, the adaptive approach will not stay converged on the actual change frequency. Hence, any adaptive approach doesn’t ensure catching all the changes; the goal is to minimize the number of fetches and maximize the number of changes detected thereby reducing the number of

network connections and concomitant data transfers. With adaptive method we can monitor the page for changes and fetch it. But issues such as how often we need to access the page and how long before we can estimate its change frequency play an important role in the performance of the adaptive method.

4.2 Page Properties

A change to a page is captured in terms of the meta-data (where available). The meta-data of the page is taken into consideration for fetching. By meta-data of the page we mean page properties such as the page size, last modified time stamp and checksum of the page. A fetch cycle for a page is triggered only when there is a change in the meta-data between the current version of the page and the previous version. Depending upon the nature (static/dynamic) of the page being monitored the complete set or subset of the meta-data is used to evaluate the change.

Static pages: For static pages, HTTP HEAD request is used to obtain the meta-data of the page. Change in time stamp of the page with an increase or decrease in page size, is flagged as change, and the page is fetched. 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. The page is cached only if the calculated checksum differs from the checksum of the previously cached copy of that page.

Dynamic pages: 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. Change is then flagged by calculating the checksum. In addition to checksum, file size is also considered to determine a change. The purpose of using the meta-data is to minimize the fetching of the page and for calculating the check sum to determine whether a page has changed. Even though the above-mentioned heuristics can be used for fetching a page based on the meta-

data we may face situations where the heuristics may fail to resolve the problem of determining if there were truly a change to the page and end up fetching the page for calculating checksum.

4.3 Determination of Fetch Interval of a Page

In cases where the last modified timestamp is available, it can be used for determination of the change frequency of that page. Generally a page is taken as changed when we find different last modified time stamp in the properties of a page between two consecutive accesses to the same page. But the change intervals thus obtained may or may not be the true intervals with which the page is changing; these intervals are what we have observed when we accessed the page. The page could have changed in the interim and therefore the last-modified date doesn't provide us with the exact frequency with which the page is changing. And this problem becomes even more difficult in case of dynamically generated pages where there are no properties associated with the page as the page is generated at the time of access. Since the fetching frequency is dependent on the change frequency of a page, we device two different approached for capturing the change to the page based on the type of the page and the information available to us on the page. In what follows we will propose an adaptive "Best Effort Algorithm" for tuning the fetching frequency to the change frequency of the page. The way we estimate the fetch frequency here based on the type of page we are handling (static/dynamic), is by capturing the history of change pattern we have observed over an interval of time. The issues are

- How much of history do we need to collect
- Replacement Policy for the captured history
- How to capture history of change pattern for dynamic pages
- Importance of recent change pattern

- How to deduce the change frequency from the collected history

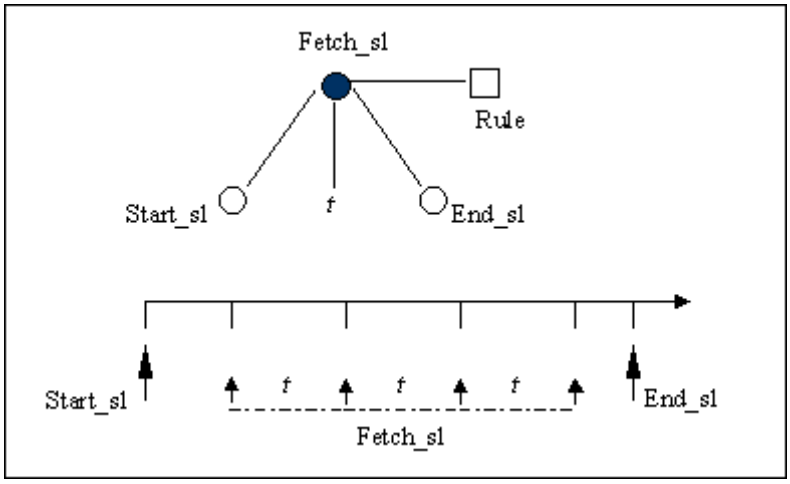


Figure 4.2 Periodic Event

4.4 Fetching

For monitoring a page, after every time interval t, we check for any change in page properties, such as the last modified time stamp of the page and then actually fetch the page if there is a change. Hence we need a mechanism for triggering the monitoring requests at a given time point after the lapse of interval t. We solve this problem by using the notion of a periodic event [19]. A periodic event is an event that repeats itself with a constant and finite amount of time.

The specification for a periodic event is *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. In WebVigiL, we use periodic events for triggering the monitoring request. 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. To actually fetch the page, we associate a fetch rule with this periodic event. Hence whenever a periodic event occurs, the rule associated with it is triggered. This fetch rule performs the functionality of the monitoring request, i.e., it fetches the page based on changes to the page properties. Consider the scenario where sentinel s1 specifies the fetch time as 2 days. The periodic event generated for s1 is as follows:

```
Event Fetch_s1 = createPeriodicEvent (Start_s1, 2days, End_s1)
```

Where Start_s1 and End_s1 are the start and end events of sentinel s1. Figure 4.2 shows the graphical interpretation of the periodic event Fetch_s1. The event Start_s1 initiates the periodic event. For every interval *t* (2 days) the periodic event is raised until event End_s1 occurs. When the periodic event is raised, the fetch rule associated with it fetches the page. For sentinels that explicitly specify the fetching interval, we generate a periodic event and associate a fetch rule with it. Hence for every unique combination of interval, start event and end event we generate a unique periodic event and associate a fetch rule with it. The interesting and difficult case is when the user expects the system to notify him/her as and when the page changes. In such cases, the system is required to tune its fetching interval with the change frequency of the page. Here we associate *one* periodic event with a fetch rule. This rule achieves the required tuning by changing the interval of periodic event. The functionality of this rule is discussed in detail in the following section.

4.5 ECA Paradigm for Fetching

Event-Based-Fetching module is responsible of monitoring and fetching pages that are of interest to the sentinels subscribed with WebVigiL. The paradigm used for fetching here is ECA, where (a) EVENT is the lapse of the fetch interval, (b) CONDITION checks for a change in the meta-data of a page, (c) and ACTION being fetching the page. We use the PERIODIC event defined in LED [19, 20]. A periodic event is defined in terms of a start_event, end_event and the interval (p) as explained in the previous section. A rule can be associated with this event. Whenever the periodic event occurs, the corresponding rule is fired, which then checks for change in metadata of the page (CONDITION), and fetches the page on change (ACTION). Thus the periodic event controls both the fetching interval and the life span of the fetch process. In the following section we define the functionality of a fetch rule and introduce two types of fetch rules namely the Best-Effort rule and Interval-Based rule based on the requirements or functionality required by the user.

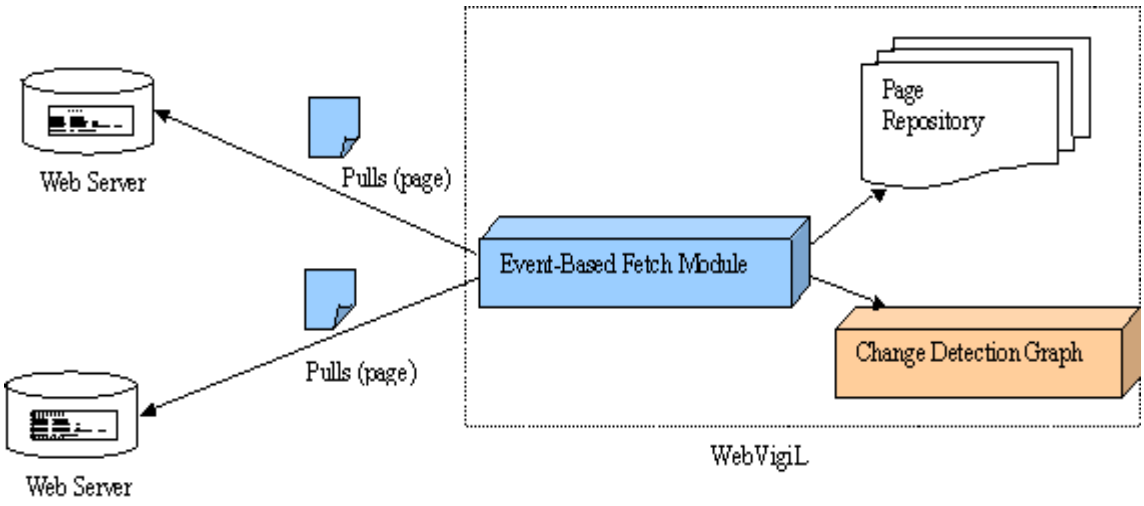


Figure 4.3 Functionality of Event-Based fetch module

4.6 Types of Fetch Rules

A fetch rule is created and used for fetching the page of interest specified in the sentinel. As per the change specification, a user can specify a sentinel with two options (a) On Change or (b) Interval Based. According to the option specified in the sentinel, Event Based Fetching module generates a Best Effort (BE) Rule or an Interval-Based (IB) Rule. These Rules differ in the way they handle the "t "(fetch interval) of the periodic event. The following table shows the E-C-A for each of the available fetch rules.

Table 4.1Types of Fetch Rules

Rule	**Event**	**Condition**	**Action**	**Comments**
Best Effort Rule	Lapse of Fetch interval	Change in meta-data of page, If change, tune Fetch Interval To observed Change Interval.	Fetch page	BE Rule tunes its fetch interval to the change frequency
Interval Based Rule	Lapse of Fetch Interval	Change in meta-data of page.	Fetch page	Fetch interval is fixed to the value set by the sentinel. Every sentinel has its own IB Rule based on the specified interval.

Best Effort (BE) Rule: In situations where the user has no information about the change frequency of a page, it is required to tune the fetch frequency to the actual change frequency of a page. The system should adapt to change frequency, ensuring the user with maximum number of changes. BE_Rule uses Best-Effort-Algorithm to achieve this tuning. Best-Effort-Algorithm (BEA) uses a history-based learning model where the next fetch interval (P_{next}) is determined from snapshot of history of changes to the page. When the next fetching interval is determined, BE_Rule changes the interval "t" of the periodic event. Clearly the performance of BE algorithm depends on the accuracy of estimation of

the fetch interval. Since there can be more than one user who is interested on a particular page, learning individually for each user would be computationally expensive.

To avoid the above mentioned scenario, Event Based Fetching Module generates a BE_Rule BEi for every unique page ui, and maps other sentinels with fetch option = “on change” on ui, to the generated rule BEi. In the earlier example, Jill specifies the fetch options as “On change” for the URL “http:// w ww. cnn. com”. As a result:

- The fetch module checks if there was a BE_Rule created for this particular URL
- If not present, it creates a BE_Rule on this URL with the start event on the StartTime of the sentinel.

Interval Based (IB) Rule: In cases where the user is interested in monitoring a page with a known frequency, the user will specify the fetch interval on the page of interest. For example Don wants to monitor a page every 4hrs starting at 9.00am, he can specify a sentinel to start monitoring the page with a fetch interval of 4hrs. So, when a user specifies an interval of interest, a periodic event whose periodicity (interval t) is equal to the given interval is created and an IB_Rule IBi is associated with it to fetch the page. As a result there will be more than one IB_Rules 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).

Clearly, one BE_Rule and many IB_Rules can be set on a common page. Thus, there can be situations where both the BE_Rule and IB_Rules fetch the same version of the page resulting in multiple copies of the same version. To avoid this situation we synchronize the fetching with the last fetched version of the same page. A rule initiates the fetch process only when, there is no version vi of the page u, with Last-Modified-Timestamp (LMT) equal to LMT of the page it is required to fetch. Figure 4.4 illustrates this problem where two IB_Rules IB2 and IB4 and a BE rule are set on the same page. If

Ti is the time of change or last modified data of the page being accessed, the figure shows the fetch process (cycle) of each rule.

As it is evident from the figure all the three rules will fetch the versions at T1, T2 and T3. Unless there is some kind of synchronization based on the version fetching or being fetched all the three rules will fetch the same version there-by leading to storage explosion at the page repository. The remedy to this situation, a version of the page is fetched by the rule only when there is no previous fetch of the version with the same last modified meta-data. In our example the fetch of IB4 and BE would be avoided as IB2 had fetched the version, they were accessing.

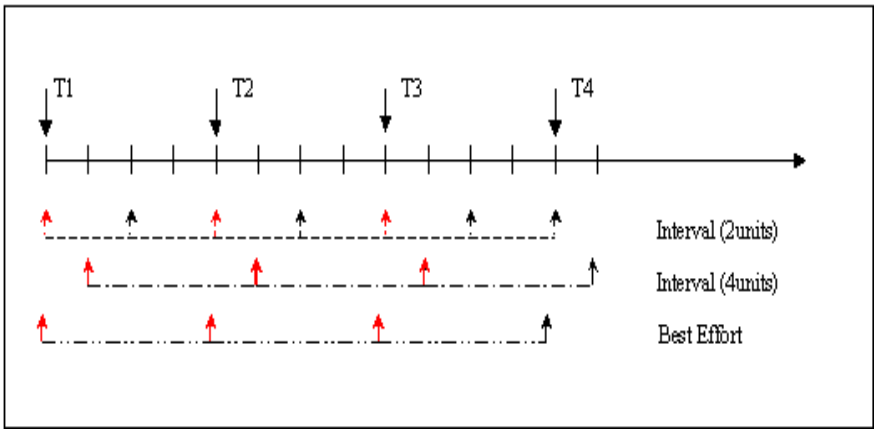


Figure 4.4 Fetch process for various rules set on a common page

4.7 Best Effort Algorithm (BEA)

Let P_{next} denote the minimum period with which the BE_Rule should poll the server so as to refresh the page if it has changed in the interim. A BE_Rule computes the P_{next} value based on observed rate of change to the page. Rapidly changing pages result in

a smaller P_{next} , whereas infrequent changes require less frequent fetches of the meta data from the server, and hence, a larger P_{next}..

Clearly, the success of the BE_Rule depends on the accurate estimation of the P_{next} value. To achieve the above objective of tuning P_{next} to the change interval of the page we take into account (a) The most recent observed intervals with which the page is changing and, (b) a static lower bound P_{min} so that P_{next} values are not set too low. Changes observed to a page can be categorized into (a) no change (b) constantly changing (i.e., changing with a constant interval), and (c) randomly changing. Prediction or calculation of the P_{next} depends on the nature of change observed during fetching. In what follows, we use $P_{current}$ to denote the interval with which the BE_Rule is polling the page currently and H_0, H_1… H_i to denote the change intervals observed during polling. P_{next} is adaptively computed as:

$$P_{next} = \begin{cases} 2*P_{current} \text{ ; No Change} \\ H_n\text{; Constant Change} \\ \text{Max } ((P_{estimate} > P_{current}\text{? } P_{estimate} * w : P_{estimate}), P_{min})\text{; Random Change} \end{cases}$$

Figure 4.5 Calculation of Pnext based on nature of change

No change: $P_{next} = 2P_{current}$, if there were no change to the page properties for α cycles, where $2 <= \alpha < t_\alpha$ where t_α is an upper bound, α is typically equal to 6 cycles. A

lower α value will result in a larger P_{next} , there by increasing the chances of missing changes in that interval.

Constant change: $P_{next} = H_n$, A page is said to be changing at a constant interval when $H_n = H_{n-1}$ .To handle situations where the actual change interval is stepped up or stepped down from that observed, we take a pessimistic approach. P_{next} is recalculated after every β cycles assuming the change to be random, $2<=\beta<t_\beta$, where t_β is the upper limit on the number cycles before we consider the change to be a constant change. Tpically β is set to 2 and is changed accordingly depending upon the constant rate. Lower values of β will result in excessive computational overhead even though the page was changing with the observed interval.

Random changes: $P_{estimate}$ with a weight factor w is used. $P_{estimate}$ is a value derived from the history of change intervals observed i.e., $P_{estimate}$ is a value computed from history, typically $P_{estimate} = \sum H_i /n$ where i=[0,n] and weight *w* initially (0.6) is a measure of the relative preference given to recent and old changes, and is adjusted by the system so that more recent changes affect the new P_{next} more than the older changes. Determination of P_{next} for pages that are varying with a random interval requires some computation. Here estimation of P_{next} not only depends on $P_{estimate}$ but also on $P_{current}$ and P_{min} . Max (($P_{estimate} > P_{current}$? $P_{estimate}$ *w : $P_{estimate}$), P_{min}) as, P_{next} value for random changes depends on the amount of history considered for the calculation of $P_{estimate}$ .A page that changed sparsely in the past and is changing at smaller intervals now, will result in having a $P_{estimate} > P_{current}$. In such cases, w is used to give more preference to new changes. P_{min} a static lower bound ensures that P_{next} will not go below a certain value. Initially, the interval of a periodic event is set to P_{min} . When the periodic event occurs, the BE_Rule is fired which uses BEA to change the interval of the periodic event to adapt to change interval. In section 4.8 we will evaluate the effectiveness of BEA against the naïve and ideal approaches.

4.7.1 Adaptability of BEA

Many factors like size of the collected history, values of α and β influence the adaptability of BEA. Below we list the effect of these factors on calculation of P_{min}.

1. Size of History: The amount of history collected for estimation of P_{next} depends on the nature of change, i.e., depending on the rate at which the page is changing correspondingly an appropriate size for the collected history should be considered. The size of the history should be a configurable parameter. In general the nature of change can be classified into fast change and slow change.

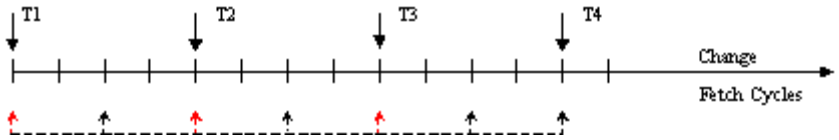
    i. Fast Change: if a page is changing at a very fast rate, irrespective of the size of the history, it will be reach its upper bound.

    ii. In cases where a page changed very fast in the past and is now changing sparsely, will result in a large stale history.

2. Value of β: Lower values of β will result in excessive computational overhead for constant changes, i.e., even when the page was changing with the observed change frequency, taking a pessimistic approach we recalculate the change frequency from the collected history to make sure the page is indeed changing with the observed frequency and larger values of β will reduce the chances of missing changes in the interim.(i.e., if the page starts changing with a lower interval)

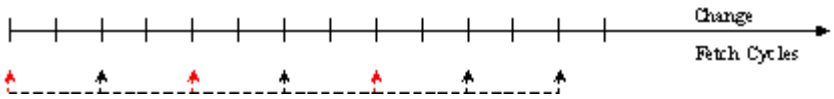
3. Value of α: This the parameter which is responsible for increasing the value of P_{next} to twice of the current polling value when a change was not observed in that many cycles. A lower α value will result in a larger P_{next} , there by increasing the chances of missing changes in that interval. It is hence advisable to have large α , it is a trade off between communication

cost vs. the need to capture a change. Lower values of α will result in lower communication cost (cycles) but increase the chances of missing changes and higher values will result in capturing all changes.



Fetch Cycle	H_i	History
1	-	-
2	-	-
3	$H_0 = (T2 - T1)$	$\{H_0\}$
4	-	$\{H_0\}$
5	$H_1 = (T3 - T2)$	$\{H_0, H_1\}$
6	-	$\{H_0, H_1\}$
7	$H_2 = (T4 - T3)$	$\{H_0, H_1, H_2\}$

Figure 4.6 Collection of History for Static Pages



Fetch Cycle	Fetch Time	H_i	History
1	T_1	-	-
2	T_2	-	-
3	T_3	$H_0 = (T_3 - T_2)$	$\{H_0\}$
4	T_4	-	$\{H_0\}$
5	T_5	$H_1 = (T_5 - T_4)$	$\{H_0, H_1\}$
6	T_6	-	$\{H_0, H_1\}$
7	T_7	$H_2 = (T_7 - T_5)$	$\{H_0, H_1, H_2\}$

Figure 4.7 Collection of History for Dynamic Pages

4.7.2 Collection Of History

It is evident that the effectiveness of the best-effort-algorithm in estimating the change frequency depends on the collected history or observed change pattern. We determine a change by looking at the page properties. Pages can be static or dynamic depending on how they are created and managed. The property that differentiates static from dynamic pages is the availability of metadata of the page. The page properties (metadata) are only available for static pages. Figure 4.6 shows the way we capture the change history based on the last modified date of the page. Between two fetch cycles if we have a two different last modified dates we could assume that page changed in the interim and thus can take the change frequency as difference in the observed time stamps. The page may or may not be changing with the captured interval, but from such intervals collected over time we can deduce a change pattern. In case of dynamically generated pages, the page is created at the page request time and hence there is no information regarding when the page changed from the previous fetch time. Hence the approach taken for static pages cannot be used for dynamically generated pages. The remedy to this problem, we use the fire time of the fetch rule as a parameter of change, i.e., if between two given fetch cycles, there were change in the meta-data of the page (i.e., check sum here) we could take the interval of change as the difference between the fire times of the fetch rules.

Even though the above approach of collecting history may not truly reflect the change pattern, we are likely to eventually catch-up with the change frequency. In what follows we will experimentally evaluate the performance of BEA against naïve and ideal approaches for static pages.

4.8 Experimental evaluation of BEA

In this section we evaluate the effectiveness of the proposed BEA against the other approaches with respect to total fetch cycles, number of times the fetch interval was tuned to the actual change interval, and the number of changes captured. We will also test the adaptability of BEA using various values of α and β and with varying history size, and experimentally deduce ideal values for theses parameters that will result in better performance of BEA. A testbed on a remote web server is used for the experiments. Test cases were generated to change a page with various change intervals. For evaluation, we categorize the test cases into (a) Fast Random (RFast#), page changing with small intervals (between 2-10 seconds) (b) Constant Change (Cons#), changing with a fixed interval (5 seconds either stepping up or stepping down with the given value) and (c) Random change (Rand#), changes randomly i.e., it may change with small intervals or constant interval or with large intervals (between 10-30 seconds).'#' Indicates the number of times a page changed. Hence the case Random240 indicates that the page changed (with a random interval) 240 times. The idea behind these test cases is to change a page at controlled intervals and to see if the proposed approach tunes the fetching frequency to the actual frequency and thereby catches maximum number of changes in minimum m number of fetch cycles.

4.8.1 Experiment CONST1000

A page on the test bed is changed 1000 times in a span of 3hrs, with interval of 5secs(stepping up or stepping down randomly). The idea is to change the page in such a way that the page changes (increasing) at a constant interval for some time and later steps

Table 4.2 Changes captured for CONST1000 with H20

TESTCASE	CHANGES	CYCLES
α2β2W.6H20CONST1000	861	1206
α4β2W.6H20CONST1000	907	1326
α6β2W.6H20CONST1000	912	1333
α4β4W.6H20CONST1000	854	1243
α4β6W.6H20CONST1000	842	1219

Table 4.3 Changes captured for CONST1000 with H50

TESTCASE	CHANGES	CYCLES
α2β2W.6H50CONST1000	829	1126
α4β2W.6H50CONST1000	869	1250
α6β2W.6H50CONST1000	849	1223
α4β4W.6H50CONST1000	847	1222
α4β6W.6H50CONST1000	813	1198

down the change interval. Table 4.2and Table 4.3show the total fetch cycles taken to capture the changes, with varying α and β values and different history sizes. It is clear form the tables that in both the cases of H20 and H50 (H# , where # is history size) α4β2 and α6β2 had better performance over the others in terms of changes detected and cycles taken to detect that many changes.

Table 4.4 Total no. of Occurrences Vs Convergence for CONST1000 H20

TESTCASE	1	0.5	0.3	0.25
α2β2W.6H20CONST1000	576	229	52	2
α4β2W.6H20CONST1000	584	239	71	11
α6β2W.6H20CONST1000	585	243	74	8
α4β4W.6H20CONST1000	550	247	46	6
α4β6W.6H20CONST1000	565	231	34	5

Table 4.5 Total no. of Occurrences Vs Convergence for CONST1000 H50

TESTCASE	1	0.5	0.3	0.25
α2β2W.6H50CONST1000	583	209	30	3
α4β2W.6H50CONST1000	566	236	61	1
α6β2W.6H50CONST1000	572	246	35	4

Convergence: Let t be the interval between two consecutive changes, we define convergence c as 1/n, where n is the number of times we polled in that interval t. Ideally, c should be equal to one. Hence the value of c lies between 0 and 1. The effectiveness of an approach in tuning the fetching interval to the change frequency can be measured in terms of number of intervals in which convergence c was 1.

With the definition of convergence established it is clear from Table 4.4and Table 4.5that α4β2 and α6β2 had more cycles of c = 1 over the rest for both cases of history sizes (H20, H50).

Table 4.6 Changes captured for RAND1000 with H20

TESTCASE	CHANGES	CYCLES
α2β2W.6H20RAND1000	936	1524
α4β2W.6H20RAND1000	971	1649
α6β2W.6H20RAND1000	978	1688
α4β4W.6H20RAND1000	922	1574
α4β6W.6H20RAND1000	917	1572

Table 4.7 Changes captured for RAND1000 with H50

TESTCASE	CHANGES	CYCLES
α2β2W.6H50RAND1000	957	1525
α4β2W.6H50RAND1000	976	1653
α6β2W.6H50RAND1000	983	1662
α4β4W.6H50RAND1000	967	1608
α4β6W.6H50RAND1000	964	1608

4.8.2 Experiment RAND1000

Clearly for constant changes α4β2 and α6β2 out performed the others in terms of changes captured and number of cycles with convergence with c = 1. Here we test for

Random changes, where the page is being changed randomly at past intervals/slow intervals or constant intervals. Table 4.6 and Table 4.7show the changes captured and cycles taken. Again α4β2 and α6β2 out performed the others for both cases of history sizes. But taking a look at the convergence factors reveals the actual effect of history size on the performance of BEA. Table 4.8 shows the convergence values for various combinations of α and β with history set to 20 and Table 4.9 for history size set to 50.

Table 4.8 Total no. of Occurrences Vs Convergence for RAND1000 H20

TESTCASE	1	0.5	0.3	0.25
α2β2W.6H20RAND1000	409	470	55	1
α4β2W.6H20RAND1000	412	490	65	1
α6β2W.6H20RAND1000	418	488	64	1
α4β4W.6H20RAND1000	359	508	45	6
α4β6W.6H20RAND1000	357	494	59	6

Table 4.9 Total no. of Occurrences Vs Convergence for RAND1000 H50

TESTCASE	1	0.5	0.3	0.25
α2β2W.6H50RAND1000	434	479	42	1
α4β2W.6H50RAND1000	381	524	65	3
α6β2W.6H50RAND1000	381	532	68	1
α4β4W.6H50RAND1000	380	542	39	3
α4β6W.6H50RAND1000	381	533	45	2

It is clear from Table 4.9 that even though α4β2W.6H50RAND1000 and α6β2W.6H50RAND1000 captured more changes, they fewer number of cycles with c= 1 when compared to when history size was set to 20 (Table 4.8). Hence taking into the consideration of the results from CONST1000 and RAND1000 experiments with H20 and H50, we prefer H20 to H50. And even though the performance of α4β2 and α6β2 for various cases was very close we choose α6β2 as it out performed the others consistently in both CONST1000 and RAND1000 experiments.

Table 4.10 Changes Captured For Various Test Cases

	RFast240		Cons240		Rand240	
	Fetch Cycles	Captured Changes	Fetch Cycles	Captured Changes	Fetch Cycles	Captured Changes
Ideal	240	240	240	240	240	240
BEA	362	206	292	239	343	224
Naive	583	240	600	240	800	240

4.8.3 Fetch Cycles vs. Changes Captured

Table 4.10 shows total number of fetch cycles taken by each approach for every test case and the number of changes captured for that many cycles. Consider the Rand240, ideally 240 changes should be detected in 240 fetch cycles. Naïve approach took 800 cycles to detect 240 changes while BEA took 343 cycles to detect 224 changes (while missing the rest). The same trend is true for all cases. So BEA captures more than 90% of changes using 40% fetch cycles as compared to the naïve algorithm. Its performance is not far from the ideal case as well.

The Y-axis of Figure 4.8 shows the total number of cycles taken by each approach for every test case and the Table 4.10 shows the number of changes captured for that

many cycles. Consider the Rand240 case, ideally 240 changes should be detected in 240 fetch cycles. Naïve approach took 800 cycles to detect 240 changes while BEA took 343 cycles to detect 206 changes (while missing the rest). So BEA ensures of capturing maximum number of changes with minimum fetch cycles.

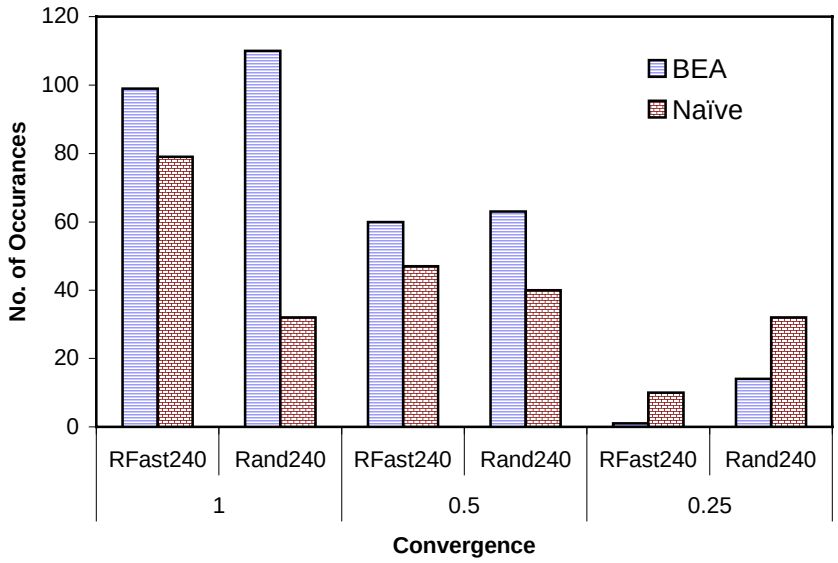


Figure 4.8 Total no. of Occurrences Vs Convergence

4.8.4 Tuning of fetch interval (Convergence)

Figure 4.8 shows the total occurrences of each convergence c of BEA and Naïve for two test cases RFast240 and Rand240. Consider the Rand240 case, where BEA polled 110 cycles with c = 1. Ideally it should have fetched 240 times. Hence 45% of the time BEA tuned its fetching frequency to the actual change frequency, where as Naïve fetched only 13% of the time with the change frequency. (Naïve fetches with a minimum frequency, without any tuning. The 13% times it polled with change frequency was mere coincidence).

CHAPTER 5

HTML CHANGE DETECTION

The World Wide Web (the Web) has become a universal repository of information and continues to grow at an astounding speed. Hyper Text Markup Language (HTML) has been used as a universal format for publishing documents on the web since 1990. In 1998, the W3C approved the eXtensible Markup Language (XML), which combined the power of SGML with the simplicity of HTML. Over the coming years XML is likely to replace HTML as the standard web publishing language but until then both will coexist. In this section, we will highlight the differences between semi-structured (XML) and unstructured (HTML) documents while emphasizing on the nature of their development. We will also introduce the problems involved in detecting changes using examples and then present the need to have different change detection models for each type of document.

In WebVigiL the change detection module detects the changes according to the user specification and notifies the presentation module. WebVigiL allows the user to specify the type of change of interest on HTML/XML pages. Given two versions, the change detection module applies change detection algorithms according to the type of documents and computes the changes based on the user profile. The changes are detected and stored in the change repository. The presentation module takes these changes and presents it in a user-friendly manner. In what follows we will discuss more on mechanisms for detecting changes to a page, and types of changes identified based on user-intent. We will present a generalize customized change detection approach termed as “CH-Diff” for detecting changes to HTML pages. At the end of this chapter we will

present various presentation schemes for showing the computed changes in a user-friendly manner.

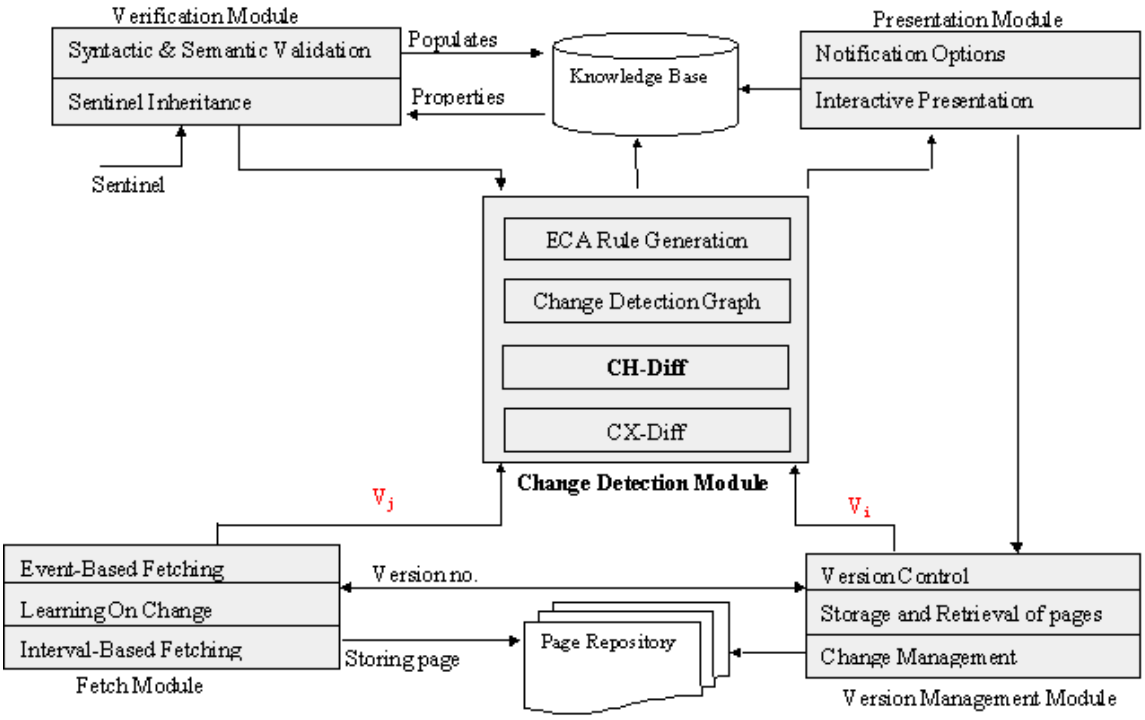


Figure 5.1 System Architecture

5.1 What is Change

It is of common interest to know changes to a document of interest on the web. Combinations of changes detected in content (data/text) and structure can flag a change to the page. There are at least three basic mechanisms for computing the change between two versions.

- **Change is flagged only when content changes**: This approach views an HTML page as a sequence of words. Presentation markups and white spaces are ignored for change computation. Therefore when a text paragraph comprised of sentences is changed into a list, no change is flagged because the content of the paragraph version matches exactly to the content of the list version, although there was a change in the presentation structure of the page.
- **Change is flagged when either content or structure changes**: This approach views an HTML page as a tree with the text (content) as leaf nodes and the tags as internal nodes. Using the true structure we can compute the change between two versions of page. But clearly in this case tree of the page containing a paragraph will be different from a tree representing a list of sentences. Thus, the example of changing a paragraph with a list will be flagged as a change, even though it was merely a presentational change.
- **Change is flagged for content changes and structure changes**: This approach can be implemented by applying above-mentioned approaches. By first approach we can detect change to content. And the second approach tells if any format change has occurred. Thus, for the example of changing a paragraph with a list will result in, no change to content but a change to formatting.

In WebVigiL, we go by the first approach i.e., we flag a change to a page only when a change in the content is detected ignoring the structural changes. Further more we assume the structure of the page to be relatively stable. A web page can be viewed as a set of markup tags and data. In an XML page, combination of the content and the tags define the nature of the content whereas in HTML they define the presentation aspects of the content. Figure 5.2 illustrates the above distinction. HTML is unstructured; but for XML the structure can be taken advantage of and can be represented as an ordered labeled tree. Hence, as the format and representation of both HTML and XML differ,

separate approaches need to be adopted for change detection for these documents. The change detection tools for HTML pages discussed in section 2.1 take into account the tags of the page along with the content for detecting change, resulting in consumption of significant computational and memory resources.

5.2 Need for User Intent

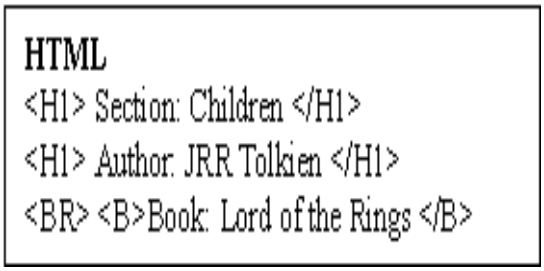


Figure 5.2 HTML Fragment

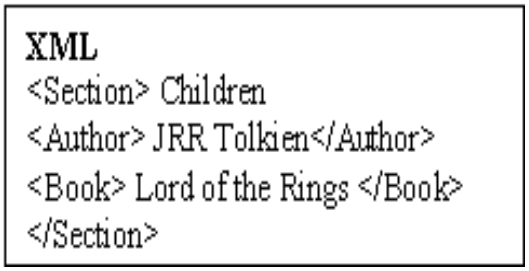


Figure 5.3 XML Fragment

The web user's interest has extended from mere viewing of information to monitoring evolution of selective information on the pages. It is therefore important to find changes to pages of interest to the user based on his preferences. These preferences can be changes to the pages (in general) or as particular as change to a fragment in the page. Hence, the change detection tool should be capable of detecting preferred changes such as appearance/disappearance of keywords, update to a phrase, etc. Consider the scenario: A student wants to monitor whether a particular course is offered. Hence the student wants to monitor the college schedule of class for that particular keyword i.e., course name. In such cases, detecting changes to the complete page is disseminating irrelevant information, leading to wasteful computation. Hence, there is a need to support detecting changes based on user's intent. Also, there should be a mechanism for a user to

specify his desired policies. WebVigiL supports customized changes and provides a user, the mechanism to define his policies. In section 5.4.1, we present a change detection mechanism for detecting customized changes to HTML pages, which not only detects changes to the entire page but also is also capable of detecting customized changes to the content.

5.3 HTML change detection

In this section we formally define change in a page, different types of changes identified and approaches for detecting these changes. In the later part of the section, we will present a generalized approach termed CH-Diff for detecting customized changes to HTML documents.

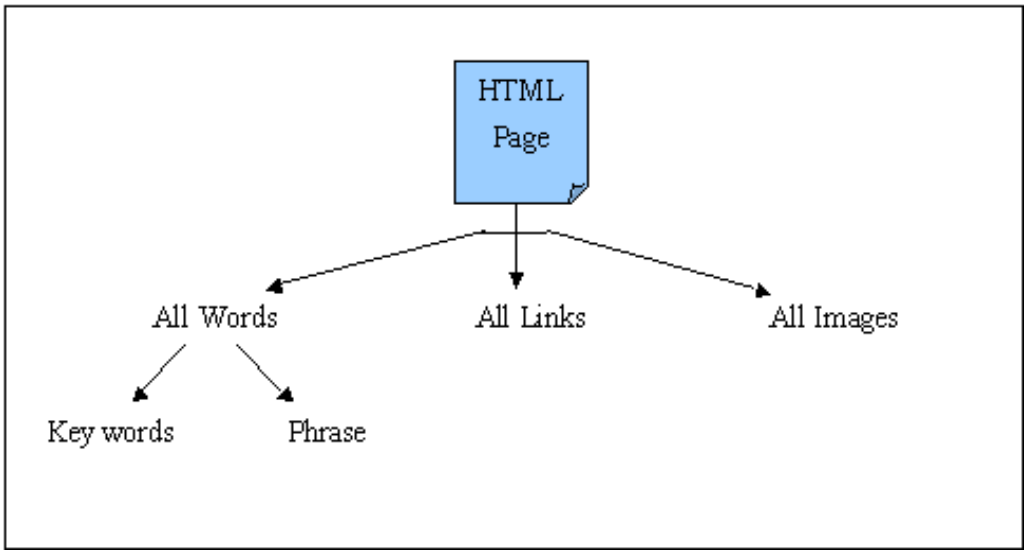


Figure 5.4 Classification of contents of a page

5.3.1 Changes of Interest in a Page

A HTML document can be viewed as a document containing raw text along with formatting and presentation markups and certain content-defining markups. As explained in the earlier section, by a change we mean change to the raw text but not to the structure of the page. In addition to detecting changes to the data between the markups, we also detect changes to certain content-defining markups, such as (<IMG src =”.”>) and (<A href =”.” >). We view a page as a sequence of words and certain content-defining markups while ignoring other markups (presentation markups). Users may be interested in the appearance or disappearance of certain words or a section of contiguous words or links/images in the page. Thus the contents of a page can be classified, based on the user intent, into keywords, phrases, all-words, links and images.

- **Keywords**: A set of unique words from the page, with no upper bound on the number of words (which are free from special characters like %$, HTML tags).
- **Phras**e: Contiguous words from the page, with no upper bound on the number of words (which are free from special characters like %$ and HTML tags). For phrases in addition to filtration of special characters we also make sure white spaces are excluded.
- **All-Words**: All the words in the page constitute this set; which also encompasses all keywords and phrases (which are free from special characters like %$ and HTML tags). All the restrictions that apply for keywords and phrases also apply to all words. In general strings with characters in the range (‘a’-‘z’,‘A’-‘Z’) or numeric range (‘0’-‘9’) are taken as valid words. (“WebVigiL”). For efficiency we may exclude articles and prepositions; in addition the user is given an option where he could provide a list of words, which can be excluded from the page during change computation.
- **Links**: A set of hypertext references (URL) extracted from the hypertext tag (<A href=”.”>). URL from tags like “mailto//” or “JavaScript://” are not considered.

- **Images**: A set of image references extracted from the image source tag (<IMG src=”.”>). Since the image source is taken as reference to the image, change in the image with the same image source reference is not taken as a change.

Since the words in phrase and keywords are a subset of the all words, we filter out the page into all-words and content-based (links, images) sets. Figure 5.5 the classification of data extracted from the page into sets of interest. We treat the context and content-tag sequence separately while keeping the processing order in the right sequence. A simple HTML parser [21] is used to parse the page and for filtering the page accordingly. Figure 5.5 shows the classification of the contents of a page based on user intent.

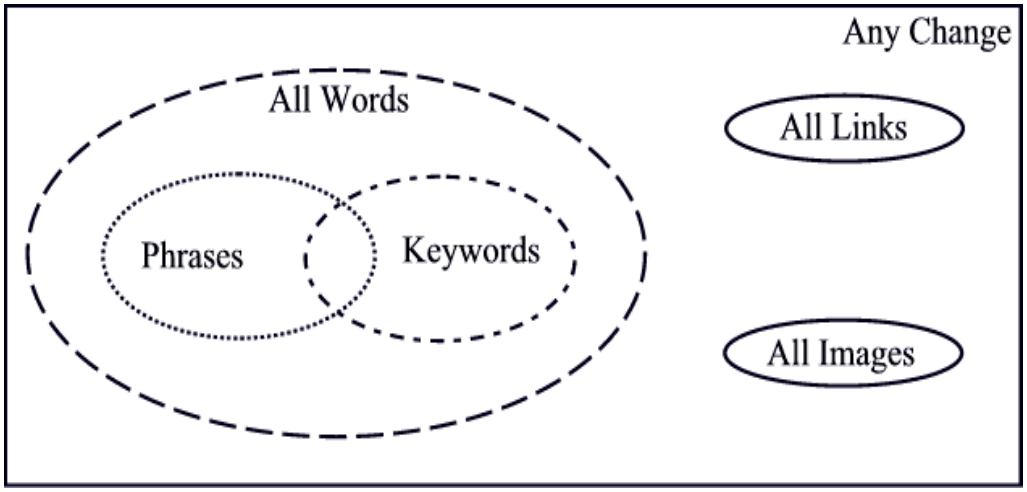


Figure 5.5 Classification of contents of a page Based on User-Intent

A user may wish to track changes at the page level or at an object level. From the user point of view, by an object level change we mean change to the object of interest to the user, such as change to a particular phrase or keyword. Hence if t is the object of interest, then set T is defined as a set of all objects of type t extracted from the page. A

page level change is any change to the page (i.e., words, links or images). For detecting changes to phrases and keywords we need to extract them from the all-words set. Object identification and extraction techniques are discussed in the following section. At the object level a change is categorized as an appearance or disappearance of an object from the page. A move of an object in the page is taken as a sequence of disappearance and appearance. In the rest of this chapter, we refer to disappearance and appearance as delete and insert respectively. In section 5.4 we will discuss more about each change type (insert/ delete/update) for the objects in a page and approaches taken for the detection of change. In general, the change detection and change presentation is achieved through three phases namely,

1. Object identification and extraction
2. Change Detection
3. Change Presentation

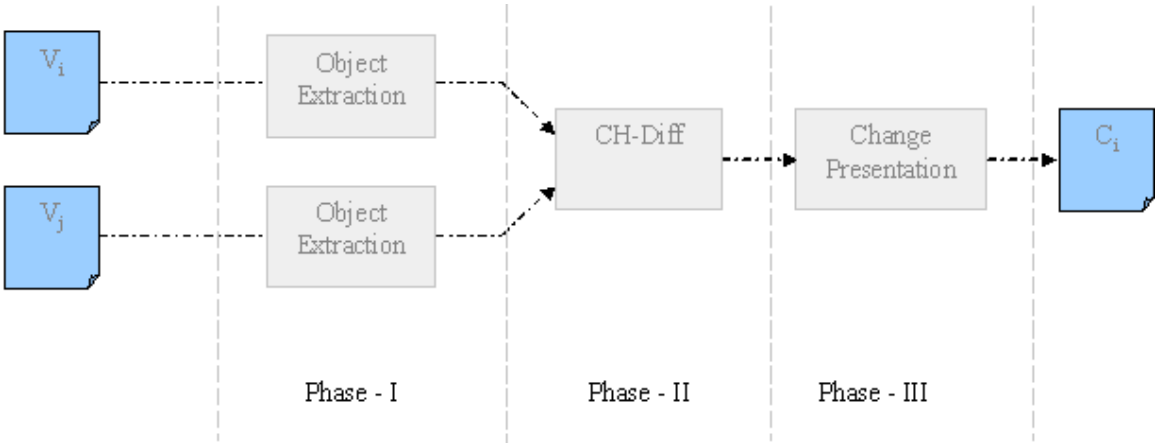


Figure 5.6 Various Phases in Change detection

5.3.2 Object Identification and Extraction

Identification and extraction of objects (links, images and keywords) from a page is straightforward whereas the task for extracting objects of type "phrase" is complicated.

Figure 5.7 A page fragment showing the signature of a phrase

5.3.2.1 Extraction of a Phrase

Consider the scenario where the phrase has been partially modified; in such a case a direct string matching will fail to locate the phrase. In order to extract the modified phrase we need additional information like the location of the phrase in the previous version. To address this problem we collect a signature to each occurrence of the phrase. The words before (lower context box) and after the phrase (upper context box) constitute the signature of the phrase. This signature form the old document is hence used for phrase identification and extraction from the new document. Consider the above figure, if "low-

risk growth investing" is the phrase of interest in the given fragment of a page then words surrounding it namely "best buys"+"Adapted from" and "long time"+"they should" define its signature. In the current prototype, we assume that the selected words surrounding the object are relatively stable. Currently, ten words (or less) before and after the phrase constitute its signature. WebCQ [16] also uses the concept of a bounding box to tackle this problem. Issues like the dynamic configuration of signature length based on the nature of the page (static/dynamic), change to signature in combination with change to phrase are being investigated.

5.3.2.2 Extraction of Image and Links

Extraction method for images and links in a page is straight-forward as each is uniquely marked in HTML page using content defining markups.

- Images are represented by tag <IMG> where the source information is specified in the URI parameter of the IMG tag, i.e. <IMG src =""></IMG>. A simple HTML parser is used for parsing the tags for IMG and accordingly the URI for the image is extracted from the source parameter.
- Links are represented by tag <A> where the hyperlink information is specified in the HREF parameter of the <A> tag, i.e. <A HREF = ""> </A>.

5.3.2.3 Extraction of Keywords

Identification and extraction of keywords is not complicated unlike phrases. As mentioned before, the page is filtered into objects of interest i.e. all-words images and links, keywords are extracted from the all-words object. Each unique keyword from the user specified list is searched for all its occurrences in all-words and a {keyword, occurrence} is generated. Thus a keyword set that is obtained from the all-words object is a collection of {keyword, occurrence}.

Change Type	Synopsis	Approach
Links	Insertion of new links or deletion of old links	CH-Diff
Images	Insertion of new images or deletion of old images	CH-Diff
Keyword(s)	Insertion or deletion of selected words	CH-Diff
Phrase(s)	Insertion/deletion/update to selected text phrase	CH-Diff
All Words	Any change to words in the page	LCS

Figure 5.8 Types of Changes with approaches taken for change detection

5.4 Detecting Changes to Objects

As discussed in section 5.3.1 the changes of interest are links, images, keywords, phrase, all-words and *anychange*. A change is captured in terms of inserts and deletes. For phrases in addition to these changes we also detect update. By update to a phrase we mean partial modification of the phrase between the versions of interest. Figure 5.8 shows when a change is flagged for each of the supported change types. By *anychange* we mean change to any word in the page and/or changes to links/images. As discussed in related work, most of the change detection algorithms/tools detect changes to a page based on the Longest Common Subsequence (LCS) where LCS for the given versions of a page gives the longest common non-contiguous subsequence for the versions. Elements which are not present in this LCS are taken as inserts/deletes according to the page being considered. Several optimizations to speedup the computation have been proposed, as calculating LCS is computationally expensive. In addition, LCS requires strictly increasing order of elements (order of occurrence) and is in effective in cases where this order of occurrence is lost. Since for detecting changes of type all-words, words of the page are extracted into the all-words object while preserving their order of occurrence we use LCS for detecting changes to words. Thus for detecting changes to all-words we use LCS. The existing tools use LCS (with several speed optimizations) to compare HTML

pages at page level. But for scenarios where the user is only interested in a change to a particular object in a page, using LCS approach will be computationally expensive. Let t be the object type, which is of interest in a page, S(A) be the set of objects of type t extracted from version V_i and S(B) be the set of objects extracted from version V_{i+1}. Here S(A) – S(B) gives the objects that are absent in S(B) indicating deletion of those objects between versions V_i and V_{i+1}. Similarly S(B) – S(A) gives the new objects that have been inserted or added into version V_{i+1}. We will improve upon this idea by introducing the concept of window-based change detection. Change detection of keywords is position independent as there is no way to keep track of word positions in the presence of multiple changes.

5.4.1 CH-Diff: Customized Change Detection Algorithm for HTML

We define a set $s\{(o_1,c_1),(o_2,c_2),(o_3,c_3),\ldots,(o_n,c_n)\}$ where $o_1,o_2,o_3..o_n$ are objects of type t with $c_1,c_2,..c_n$ being the corresponding number of occurrences (> 0) of each object in a version V_i. For detecting changes to objects of type t in version V_i, we need to compare the set obtained from V_i, with the old set obtained from version V_{i-1}. Increase or decrease in the number of instances of an object, is taken as an insert or delete. As the name of the approach indicates we form a window of objects which are ordered based on their hash codes. In java 1.3, for every string object a hash code [22] is generated (since every object is of type string in our case we use this hash code). The hash code of the first and last object in the re-ordered old set defines the bounds of the window.

- **Phase-I**: Every object in the new set with a hash code greater than the upper bound or lower than the lowerbound is flagged as an insert. Objects that have their hash code within [lowerbound, upperbound] are searched for occurrence in the objects defining this range (in re-ordered old set). If found, the occurrence count is compared and

accordingly an insert or delete is flagged. If not found then the object in the new set is flagged as insert.

- **Phase-II**: All objects in re-ordered old set that have not participated in the previous phase are flagged as deletes.

Currently since the objects are extracted from the page and change is deduced from the occurrence count, we cannot know exactly which instance of this object changed in the page. The change flagged after the above two phases can be classified as changes that are unique (unique inserts and unique deletes) and changes to objects that have multiple instances. Thus to differentiate the above we categorize changes as

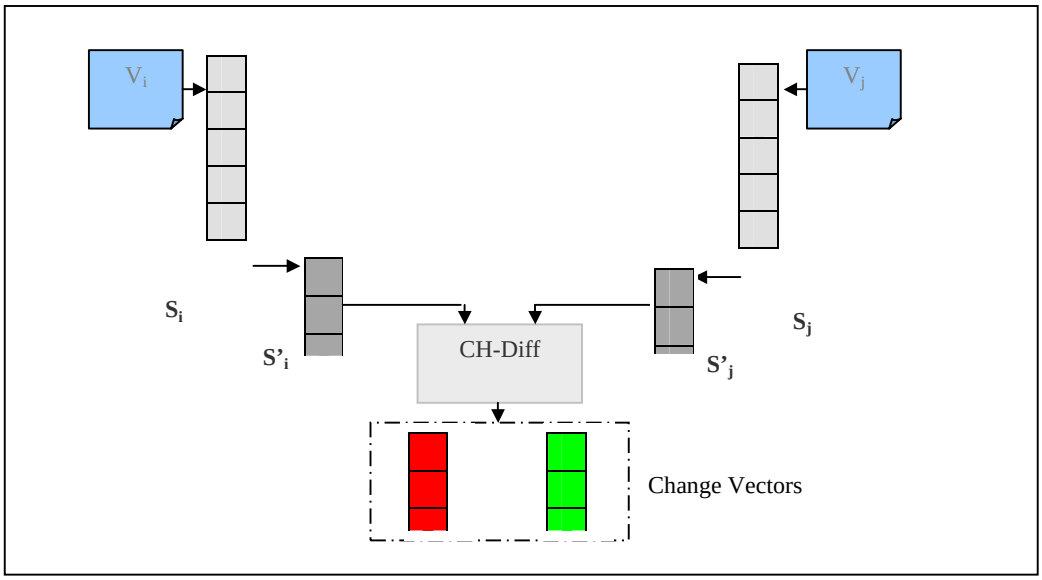


Figure 5.9 Shows the sets obtained between various phases

- "i" : unique insert
- "d" : unique delete
- "i+"
- "d-"

Objects that have been marked, as “i+” and “d-” require additional computation to determine which instance of the object had actually changed. We do this additional computation only for phrases.

5.4.1.1 Phrase

For phrases, in addition to insert and delete, an update to a phrase is also detected. Here, for phrases, the objects in the extracted set denote the signature of each instance (occurrence) of the phrase. The process involved in change detection of a phrase is as follows:

1. Initially during the object extraction phase, Knuth-Morris-Pratt (KMP) string-matching algorithm is used for matching of the phrase with the words extracted from the page (words object).

2. For every hit, the corresponding signature (bounding box) is extracted. Thus the set of objects extracted from the new and old version of a page for phrase detection is the signature of each instance of the phrase in the corresponding version.

3. The window-based approach results in indicating inserts and deletes to the objects considered. Delete to the object here has two possibilities: Either the phrase was completely deleted from the new version or the phrase is partially updated but is not caught by KMP. Here we use a heuristic approach for determining an update to a phrase.

4. The object that was flagged as deleted in the old set (i.e., signature of that instance of a phrase) is used against the new page to get the words wrapped by this signature.

5. LCS is now run on the words thus extracted and the phrase, and if the length of the resulting longest common subsequence is greater than a given percentage (the percentage can be adjusted on the fly based on past history) of the length of the phrase we take it as an update else a delete is flagged.

```
Input (Page P1, Page P2, User given Phrase)
/*Parsing the page and extraction of words from the page */
1.  Parse and Extract the words from P1 into Old_Set
2.  Parse and Extract the words from P2 into New_Set

/*Extraction of signature of each instance of phrase from the words sets */
3.  Extract the signature(phrase) from Old_Set into set K1
4.  Extract the signature(phrase) from New_Set into set K2

/*Sort the set based on the hashCode of its elements forming a range [lowerbound, upperbound] */
5.  Sort K1 based on hash code
6.  lowerbound  =  hashCode (the first element in K1)
7.  upperbound =  hashCode (the last element in K1)

/*Finding inserts and deletes*/
8.  for every element l in k2
10.   if (hashCode(l) > outerbound || hashCode(l)  < lowerbound )  append {l} to insert _set
12.   if (hashCode(l)  <= outerbound || hashCode(l)  >= lowerbound )
16.      if (l exists in K1)no change
17.      else append {l} to insert_set.
18. for every l in K1 that didn't  participate in the previous phase, append {l} to delete_set

/*Detecting Updates*/
19. for every l in delete_set
20.   get lower_context_box = lower_context_box(l)
21.   get upper_context_box = upper_context_box(l)
22.   updatecontent = ExtractContentBetween(lower_context_box,upper_context_box,Old_Set)
23.   Length = LCS(phrase,updatecontent)
24.   If (Length /phrase.length < updateFactor ) flag update
25.   Else flag delete
```

Figure 5.10 Outline of the CH-Diff Algorithm for phrase

Consider the example where a user is interested in monitoring changes to objects {a, b, c, d, e} and let the set instances of these objects extracted from a old and new version of a page be {(d,2),(b,2),(c,2)} and {(a,1), (c,2),(b,2),(e,1)} respectively. By looking at these sets its obvious that objects “a” and “e” have been inserted in the page and both the instances of object “d” are deleted from the old page. We will now deduce these results using our approach of change detection.

Phase-I: The old set {(d,2), (b,2), (c,2)} is reordered based on the hash code of each object resulting the set {(b,2) ,(c,2), (d,2)}. The hash code of “b” and “d” form the lower and upper bound for the collection of objects in old set thus defining the window (step 1). Now hash code of every object in the new set is compared against the bounds of the window. Since the hash code of “a” is lower than the lowerbound (step 2) it is added into the insert list. Similarly “e” is added to the insert list as its hash code is greater than the upperbound (step 5). Objects, which occur with in the bounds, are searched in the old set. Since the occurrence count of “b” and “c” were unchanged, they are added into an unchanged set.

		Re-ordered Old Set	Object in New Set	Insert set	Delete set	No change set
Phase-I	1.	{(b,2), (c,2),(d,2)}	(a,1)	{}	{}	{}
	2.	{(b,2), (c,2),(d,2)}	(a,1)	{(a,1)}	{}	{}
	3.	{(b,2), (c,2),(d,2)}	(c,2)	{(a,1)}	{}	{(c,2)}
	4.	{(b,2), (c,2),(d,2)}	(b,2)	{(a,1)}	{}	{(c,2),(b,2)}
	5.	{(b,2), (c,2),(d,2)}	(e,1)	{(a,1),(e,1)}	{}	{(c,2),(b,2)}

		Re-ordered Old Set	New Set	Insert set	Delete set	No change set
Phase-II	1.	{(b,2), (c,2),(d,2)}	{(a,1), (c,2),(b,2),(e,1)}	{(a,1),(e,1)}	{(d,2)}	{(c,2),(b,2)}

Figure 5.11 Operations in Phase-I and II

Phase-II: Once we reach this phase only those objects that are completely deleted in the new version are not detected. For determining this we use the no change set, where, the objects not in this set but in the old set indicate the complete deletion case. Hence in the current example (d,2) which is not present in the no change set is taken as deleted and added into the delete set.

The change computed (i.e., the insert set and delete sets,) is used in change presentation phase for presenting the changes. In following section we will discuss about presentation schemes and point out the type of scheme taken for each change type.

5.5 Change Presentation

Change presentation is the last phase of web monitoring where the detected changes, as outlined in the previous sections, are presented to the user. For meaningful interpretation of the presented changes, we have investigated three ways to present it to the user:

- **Only Change Approach**: Showing only the changes and omitting the common objects of the two pages is advantageous for pages of large size but will make interpretation intricate. This approach can be meaningful for hand-held devises to conserve the amount of data transmitted over a limited bandwidth.
- **Single Frame Approach**: Produce a single document by merging the two documents summarizing all inserted, deleted and common objects. The advantage lies in displaying the common objects just once, but with the draw back of possibly changing the page structure.

- **Dual Frame Approach**: Showing both the documents side-by-side in different frames and highlighting the changes between the documents has the advantage of uncomplicated interpretation of the changes presented. When the number of changes to be presented is large, this approach may make it difficult to interpret the changes. This can be remedied by presenting parts of the pages at a time to limit the number of changes displayed in each installment.

In WebVigiL we intend to use all of the three presentation schemes summarized above in a selected combination depending upon the type of change type being presented. For example we plan on testing

1. The dual frame approach for presenting changes to phrases and keywords.
2. The single frame approach for displaying changes to image and links (showing both the old and new image).
3. A heuristics-based approach, for the change type any-change, based on the number of changes detected; we use a heuristic cost model for choosing the presentation mechanism between the Dual Frame Approach, Only Change and single Frame Approach for displaying changes.

An example of WebVigiL's Dual Frame output is shown in Figure 5.14 for the given keywords {CSE2315, CSE2320, ALGOR&DATASTRUC, CSE3310}. Markups are used to highlight deleted and inserted objects. Deleted text is displayed in "struck-out" font using <STRIKE><B> which, as experimentally determined in [5], is rarely used in HTML and XML documents. And for displaying inserted text, we are currently using colors and <I><B> to highlight.

Currently modified "content-defining" markups like images and links we present changes in a tabular format, indicating the insertions or deletions. For keywords we produce the Dual Frame, only highlighting unique inserts and deletes.

Entry	OldCount	NewCount	Change
"http://us.i1.yimg.com/us.yimg.com/i/mntl/re/yre2.gif"	1	-1	Delete
"http://us.i1.yimg.com/us.yimg.com/i/spo/bballdot.gif"	3	2	Delete
"http://us.i1.yimg.com/us.yimg.com/i/mntl/re/yre3.gif"	-1	1	Insert

Figure 5.12 Presentation for image change.

Entry	OldCount	NewCount	Change
"http://www.messenger.yahoo.com"	1	-1	Delete
"http://www.auctions.yahoo.com"	1	1	--
"http://www.cnn.com"	1	1	--
"http://www.yahoo.com"	1	1	--
"http://www.maill.yahoo.com"	1	1	--

Figure 5.13 Presentation for links change.

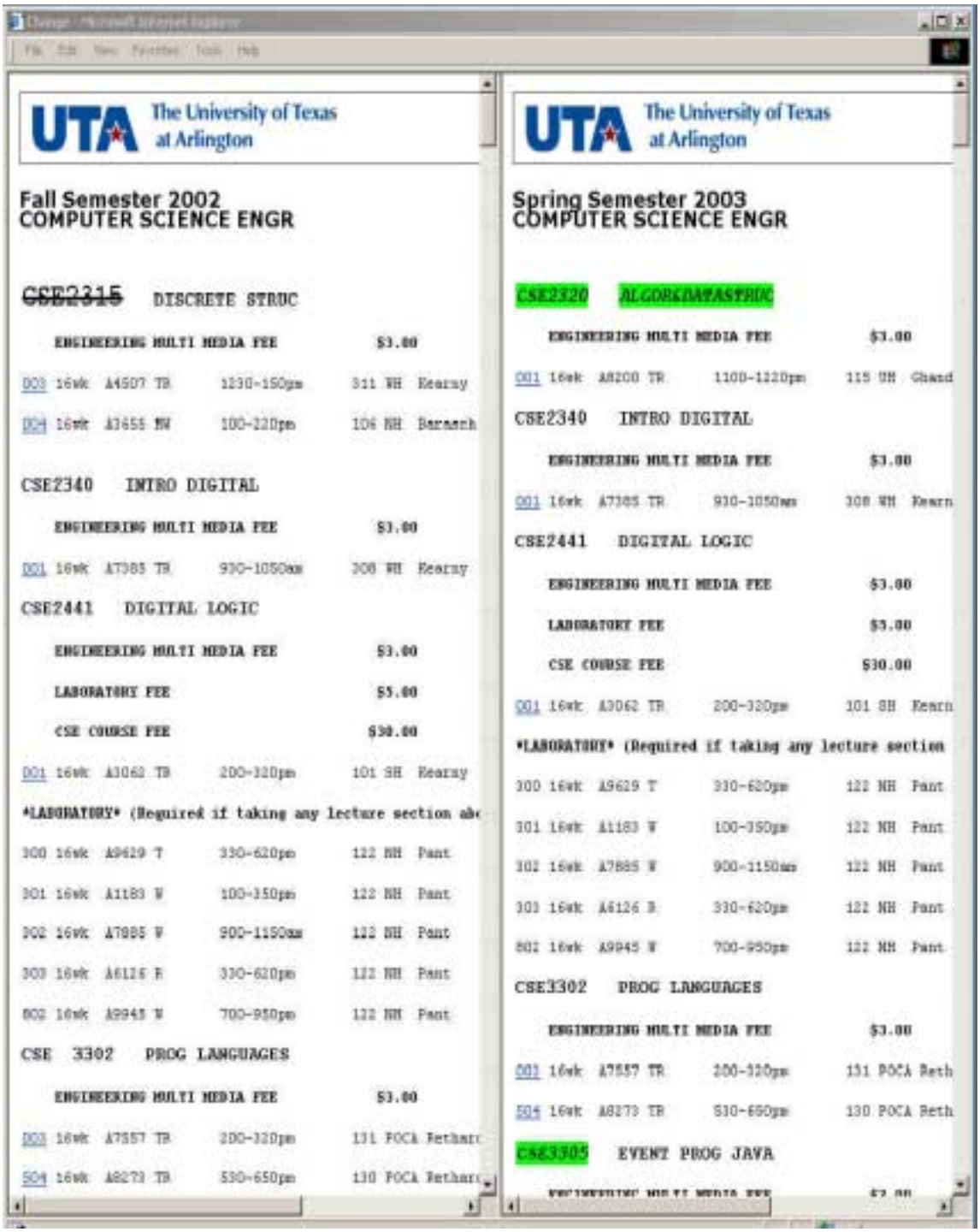


Figure 5.14: Change Presentation for keywords

CHAPTER 6

IMPLEMENTATION OF FETCHING

In this chapter we discuss the implementation details of the fetching mechanisms. The structure of this chapter is as follows: we first go into the details of API's provided in java 1.3 for making connection and retrieving properties of pages and we discuss in detail how each property of meta-data of a page is retrieved using the existing API's. We will also present implementation details on how the API's are used for page fetching. Implementation of various fetch rules and implementation differences of the rules is presented in detail in the later part. Examples of rule creation are provided for better understanding of the functionality of the fetch rules. We present implementation details of the Best-Effort-Algorithm (BEA) for calculating fetch interval for various observed change patterns. We end the chapter by presenting the details on the Testbed that was used for various experiments for evaluating the performance of BEA algorithm.

6.1 Implementation of Fetching Page Properties

In this section we will discuss the API's used for fetching the page properties and the page content itself. Uniform Resource Locator, is a pointer to a "resource" on the World Wide Web. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine. In general, a Url can be broken into several parts. Consider the example "http:// i t lab.uta.edu/ s harma/ projects/Web VigiL/ index. html". Here the Url indicates that the protocol to use is http (Hypertext Transfer Protocol) and that the information resides on a

host machine named itlab.uta.edu. The information on that host machine is named /sharma/projects/WebVigiL/index.html. The exact meaning of this name on the host machine is both protocol dependent and host dependent. The information normally resides in a file, but it could be generated on the fly (dynamically generated at query time). This component of the Url is called the *path* component. The syntax of Url is defined by *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax,* amended by *RFC 2732: Format for Literal IPv6 Addresses in URLs.*

In WebVigiL, we fetch a page only when there is change in its page properties like last modified data, file size or checksum. Hence given a Url, it is required to first retrieve the page properties and if change is observed we need to fetch the contents of the page. The class URL represents a Uniform Resource Locator whose member functions provide information regarding the remote file pointed by the Uniform Resource Locator.

```
URL url  = new
URL("http:// i t lab.uta.edu/ s harma/ projects/Web VigiL/ index. html");
```

The abstract class URLConnection is the superclass of all classes that represent a communication link between the application and a URL. Instances of this class can be used both to read from and to write to the resource referenced by the URL.

```
URLConnection urlConnection = new  URLConnection(url);
```

In general, creating a connection to a URL is a multistep process:

1. Invoking the openConnection method on a URL creates the connection object.

2. The setup parameters and general request properties are manipulated.

3. The actual connection to the remote object is made, using the connect method.

4. The remote object becomes available. The header fields and the contents of the remote object can be accessed.

The following methods are used to access the header fields and the contents after the connection is made to the remote object:

getContentEncoding

getContentLength

getContentType

getDate

getExpiration

getLastModifed

The getContentType method is used by the getContent method to determine the type of the remote object; for example we distinguish between XML and HTML files by looking at this file, for XML files this method return “text/XML” and for HTML files “test/HTML”.

```
urlConnector         = url.openConnection();
int contentLength    = urlConnection.getContentLength();
String contentType = urlConnection.getContentType();
```

For fetching the contents of the URL, we use openStream() method of URL class. It opens a connection to the given URL and returns an InputStream for reading from that connection. This method is a shorthand for: openConnection().getInputStream(). An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and

decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted. In our case we use the default charset.

```
InputStream in = urlHandler.openStream();
```

Each invocation of one of an InputStreamReader's read() methods causes one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation. For efficiency, we wrap an InputStreamReader within a BufferedReader. For example:

```
BufferedReader reader  = new BufferedReader(new InputStreamReader(in));
```

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

```
while ((buffer = reader.readLine()) ! = null ) {…….. }
in.close();
```

close() method of the InputStreamReader closes the established connection to the remote file, when all the data from the specified file is read using the BufferedReader. In cases where checksum of the retrieved page is to be calculated, the checksum is computed after the data from the remote file is buffered. Java1.3 provides a class CRC32 with methods for calculating checksum.

```
import java.util.zip.CRC32;
CRC32 checksum = new CRC32();
checksum.update(buf.toString().getBytes());
checksum.getValue();
```

In general to retrieve the header information of the URL, we create an instance of URL class with the given Url as the parameter to it, and use this instance as a parameter for creating an instance of URLConnection, whose member functions can be used for checking the page properties. For fetching the content of the given page (Url), we open a connection using the URL instance, and read the data by buffering. The following tables show the member function of URL and URLConnection classes along with a brief description of functionally.

Table 6.1Methods used for opening connections and streams in URLConnection class.

Method	Return type	Description
OpenConnection	URLConnection	Returns a URLConnection object that represents a connection to the remote object referred to by the URL.
OpenStream	InputStream	Opens a connection to this URL and returns an InputStream for reading from that connection.

Table 6.2 Methods in class URL used for fetching.

Method	Return type	Description
getContentLength	int	Returns the value of the content-length header field (file size).
getContentType	String	Returns the value of the content-type header field (type/document)
getLastModified	Long	Returns the value of the last-modified header field.

6.2 Fetch Rule Creation and Initialization

In this section we will discuss the implementation details of creation of periodic events, associating fetch rules with the periodic events. In WebVigiL we have two types of fetch rules Interval-Based and Best-Effort which are distinguished based on how they treat the interval, with which the periodic event occurs.

The paradigm used for fetching here is E-C-A. 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 paradigm has been used for monitoring the database state in active databases and as a stand-alone concept for monitoring objects in applications (both centralized and distributed [23]). As part of the Active Object-oriented system [24, 25], a 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. ECA rules provide an elegant mechanism for supporting asynchronous executions based on events (temporal or otherwise).

ECAAgent class contains the API to be used for Sentinel applications. The ECAAgent instance stores the names of all events and their associated event handles. With this API, the user can create class level and instance level Primitive events, Composite events and define rules on those events in different parameter contexts (RECENT, CHRONICLE, CONTINUOUS, CUMULATIVE). The user also raises the events through the 'raiseEvent' API. This class is a public class since it is accessed by all user applications.

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

The specification for a periodic event (composite event) is *PeriodicEvent (E1, [t], E3)* where E1 and E3 are the events (or time specifications) that act as an initiator and a terminator, and *t* is the time interval. Consider the following example, where start and end events, which are primitive events, are raised when a particular method of a class is called.

```
EventHandle start = myAgent.createPrimitiveEvent("start_"+sURLName,
                                    "pushpull.BestEffortFetchPageContent",
                                     EventModifier.BEGIN,
                                    "public void startEvtPageChecking(String
                                     sURLName)",this);
EventHandle end = myAgent.createPrimitiveEvent("end_"+sURLName,
                                   "pushpull.BestEffortFetchPageContent",
                                   EventModifier.BEGIN, "public void
```

endEvtPageChecking()",**this**);

A periodic event (composite event) can now be declared over the two events start and end. EventHandle exp = myAgent.createCompositeEvent(EventType.PERIODIC,

"Expression_"+sURLName,start,sTimeString,this,end);

Here the periodic event is raised after every interval “sTimeString” once the start event is raised till the end event is raised. Periodic event is of type composite event, the following shows the parameters taken in creation of a composite event.

```
createCompositeEvent(EventType eventType,
                     java.lang.String eventName,
                     EventHandle leftEvent,
                     java.lang.String timeString,
                     java.lang.Object instance,
                     EventHandle rightEvent)
```

Once a periodic event is declared, rules can be associated with it, which are invoked every time the periodic event is fired. The syntax of createRule Method that is used for creating a rule is show 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. Consider the following example, where a rule is associated with a periodic event exp.

myAgent.createRule("Rule"+sURLName,**exp**,

"pushpull.BestEffortFetchPageContent.checkCondition",

"pushpull.BestEffortFetchPageContent.performAction");

Consider the following class BestEffortFetchPageContent that is a member of the package pushpull. As described above start and end events were declared on the invocation of startEvtPageChecking and endEvtPageChecking functions of a particular instance of this class. The checkCondition and performAction functions are called as per the E-C-A paradigm by the rule defined over the periodic event. Depending on the result (true/false) from the condition part the action is performed, which in our case is fetching of contents of a given page.

```
Package pushpull;
.
.
public class BestEffortFetchPageContent { ..
        public boolean checkCondition(ListOfParameterLists paramList){. .}
        public void performAction(ListOfParameterLists paramList){. .}
}
```

```
public boolean checkCondition(ListOfParameterLists paramList){
        page = new PageProperties();
        page.openLink(sURLName);
        currentModified = page.getPageLastModified();
        // if the page is dynamically generated or has no page properties
        if(currentModified == 0){
        currentModified = System.currentTimeMillis();
        .
        changeInterval = currentModified - lastModified;
        .
        lastModified = currentModified;
        .
        newCheckSum = page.getChecksum(page.getPageText());
        if(newCheckSum != oldCheckSum){
                oldCheckSum = newCheckSum;
                change = true;
                history.addElement(new Long(changeInterval));
        }…
        pLearned =  PTune.frequencyLearning(change,pCurrent,history);
        if(pLearned != pCurrent)
myAgent.setPeriodicEventTimeString(startEvtPageChecking[0],
                                TimeChange.getHHMMSS(pCurrent));
}//checkCondition
```

In the checkCondition method, checksum is calculated for pages that have no page properties or generated dynamically. In these cases the last modified timestamp of the remote page will be zero. Best-Effort and Interval-Based Rules differ in the condition part. BE Rules depending upon the estimated value of change frequency, change the interval of the periodic event. Where as the Interval Based rules don't collect the change history and fetches the page at specified interval on change.

Table 6.3 Various Methods of class ECAAgent used in creation of events and rules.

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 creates an instance level rule on the specified object instance
setPeriodicEventTimeString	EventHandle	This method changes the interval of the periodic event.
ECAAgent.insert	static void	This method inserts a char parameter into the parameter list of the event handle.
ECAAgent.raiseBeginEvent	static void	This method raises an event at the beginning of a method.
ECAAgent.raiseEndEvent	static void	This method raises an event at the end of a method.

Table 6.4 Methods that constitute the rule body.

Method	Return Type	Description
checkCondition	Boolean	This method checks for change in page properties.
performAction	Void	This method fetches the page.

6.2.1 Fetch Content Properties File

The condition part of the fetch rule is the place where we capture the history of the changes occurring to the remote page. There is a bound on the amount of history captured, i.e. as the size of the history increases over the bound, the past history is axed. This bound on the size of captured history is a configurable parameter specified in the fetch content properties file. Along with the size, two more properties are provided that can be used by the developers for debugging and to log the process of fetching. These flags are DEBUG [true/false] and LOG [true/false].

Table 6.5 Classes provided in pushpull package

Class	**Description**
BestEffortFetchPageContent	The fetch interval is tuned in the condition part
IntervalBasedFetchPageContent	The fetch interval is not tuned in the condition part

6.3 Implementation Of Best-Effort-Algorithm (BEA)

Best-Effort-Algorithm (BEA) uses a history-based learning model where the next fetch interval (P_{next}) is determined from the history of changes to the page. The condition part of fetch rule is the place where the history is accumulated. When the size of the accumulated history exceeds the limit set in the fetch content properties file, the past history is axed. Difference in the last modified timestamps is taken as the observed change interval and is accumulated as history of change. Here we use the BEA for learning frequency of fetching based on the history. The following shows the control flow in the condition part of the rule.

```
//pCurrent is the current fetching or polling interval
currentModified = page.getPageLastModified();
if(currentModified != lastModified){
history.addElement(currentModified- lastModified);
change = true;
// Replacement of History
 if (history.size()== historySize)   history.removeElementAt(1);
}
FrequencyTunning PTune = new FrequencyTunning();
pLearned =  PTune.frequencyLearning(change,pCurrent,history);
```

FrequencyTunning class provides the API required for tuning the fetching frequency. It provides the implementation of BEA. FrequencyTunning is responsible for calculating the next fetching interval based on the change i.e., no change, constant and random change. It uses a function for estimating using history and the current fetch interval. If there were no change for α cycles then the current fetch (polling) interval is

doubled and returned as the next interval. If there was a change and the change was constant (last two entries in the history were same) the current history entry is retuned and after β cycles, the interval is recomputed as if the change was a random change (last two entries into the history are not same).

```
//alpha , beta are set to default values set in the config file.
public long frequencyLearning(boolean changeType, long Pcurrent, Vector H){
        // no change
        if(!changeType){
 alpha--;
        if( alpha ==0){
                Pcurrent = 2*Pcurrent;
              }
        }//no change
        if(changeType){
                // constant changes
                 if(HN == HN_1){
                     if(constChangeCycles == beta){
              Pcurrent = estimate(H); ……
                     }
                 }
```

```
                    // random change
                    Pcurrent = estimate(H);
            }//change
            return Pcurrent}// frequencyLearning
    }// frequencyLearning

    long estimate( Vector H){
    meanHistory = averagingHistry(H);
    nextVal = Max(meanHistory>=Pcurrent ? meanHistory * w ): meanHistory,
     Pmin);
    return nextVal;
    }// estimate
```

α, β and w that are used in frequencyLearning function are set to default values during the class initialization time from the PollingProperties file. A testbed on a remote webserver was used for the experiments. Test cases were developed to change a page with various change intervals.

CHAPTER 7

IMPLEMENTATION OF CH-DIFF

In this chapter we discuss the implementation details of CH-Diff: Customized Change Detection of HTML pages. The structure of this chapter is as follows: we start with implementation details of HTMLParser used to parse the HTML page for data extraction, we will discuss how the existing features provided by the parser were modified for our needs, implementation details of CH-Diff in detecting changes to links, images, keywords, phrases is presented in the later half. We conclude the chapter by providing the details on testhandles used for validating each detection mechanism and finally we present the implementation details of the presentation of links, images and keywords change.

7.1 Quiotix HTMLParser

Quiotix HTMLParser[21] is a JavaCC grammar for parsing HTML documents. It builds a simple parse tree, which is used to validate, reformat, display, analyze, or edit the HTML document. The parser produces a parse tree that throws away very little information contained in the source file, and hence dumping of the parse tree results in an almost identical copy of the input document. The source information that is discarded by the parser is whitespace inside of tags (i.e., the spaces or new lines between the attributes of a tag.) The generated parse tree supports the commonly used "Visitor" design pattern. Several visitor classes have been provided, which do things like dump the parse tree, restructure the parse tree, etc. Common tasks such as formatting, validation, or analysis are easily performed as Visitors.

Table 7.1Classes in the webvigil.Hparser used for parse HTML page.

Class	**Description**
HtmlParse	Generates the parse tree
HtmlDocument	Represents an HTML document as a sequence of elements.
HtmlCollector	An HtmlVisitor which modifies the structure of the document so that begin tags are matched properly with end tags and placed in TagBlock elements.
HtmlScrubber	HtmlScrubber is a Visitor which walks an HtmlDocument and cleans it up.
HtmlVisitor	Abstract class implementing Visitor pattern for HtmlDocument objects.

The parser transforms an input stream into a parse tree; the elements of the parse tree are defined in HtmlDocument. One can traverse the tree using the Visitor pattern; the base visitor is defined in HtmlVisitor, and there are several visitors, which are part of the HtmlParser package. HtmlDumper is a simple visitor, which traverses the parse tree and reconstructs the original document and writes it to System.out. HtmlCollector and HtmlScrubber are more sophisticated visitors, which transform the parse tree. HtmlCollector imparts a tree structure to the otherwise flat parse tree, matching begin and end blocks with each other.

HtmlScrubber cleans up the documents, converting tags and attributes to upper or lower case, removes unnecessary quotes and white space, etc. To parse a document, we invoke the HtmlDocument method on the parser. This will produce an HtmlDocument, which is a sequence of HtmlElement objects. HtmlElement include subclasses Tag,

EndTag, Comment, Text, Newline, and TagBlock (TagBlock is a composite object comprising of a tag, a matching end tag, and a sequence of the intervening elements.) The Parser does not attempt to match start tags with end tags (so the result of the parsing process will not contain TagBlock elements.) instead uses the a bottom-up parsing mechanism, the HtmlCollector class (a subclass of HtmlVisitor) walks the document and attempts to match up tags and impart more structure to the document. HtmlDocument class represents an HTML document as a sequence of elements. The defined element types are: Tag, EndTag, TagBlock, Comment, Text, Newline, and Annotation. The various element types are defined as subclasses of HtmlElement and are static for efficient access.

Table 7.2 Static subclasses of HtmlDocument.

Class	**Description**
HtmlDocument.HtmlElement	Abstract class for HTML elements.
HtmlDocument.Text	Plain text.
HtmlDocument.Tag	HTML start tag.
HtmlDocument.TagBlock	A tag block is a composite structure consisting of a start tag a sequence of HTML elements, and a matching end tag.
HtmlDocument. acceptLinks	Extracts all links in the page
HtmlDocument. acceptImages	Extracts all images in the page
HtmlDocument. acceptWords	Extracts all words in the page(filters special characters)

HtmlScrubber is a Visitor, which walks an HtmlDocument and cleans it up. It can change tags and tag attributes to uppercase or lowercase, strip out unnecessary quotes

from attribute values, and strip trailing spaces before a newline. Here we capture the data required i.e., allwords, links and images. We have modified the following member functions for capturing the required data:

- For capturing words in the page (text between the tags)

```
public void visit(HtmlDocument.Text t) {
 if(start && !stop){    // for extracting data only between <body>..</body> tags
        .
        t.text = notLetterOrDigit(t.text); // eliminating special characters
        if (!(t.text.equals("nbsp")))        // eliminating nbsp tag in HTML
              allWords.append(t.text);   // capturing the data
}//visit
```

- For capturing links and images in a page

```
public void visit(HtmlDocument.Tag t) {
 if( t.tagName.equals("body") ) start = true; // this is for <body> tag
 if(start && !stop) // for extracting data only between <body>..</body> tags
 {
        if(b.startsWith("HREF") ){    allLinks += b+";";
        if(b.startsWith("SRC") ){      allImages += b+";";
}//visit
public void visit(HtmlDocument.EndTag t) {
if( t.tagName.equals("body") )  stop = true; // this is for </body> tag
```

Here allLinks, allImages, allWords are String Buffers that are declared in HtmlVisitor class. HtmlScrubber extends HtmlVisitor, which is an abstract class implementing Visitor pattern for HtmlDocument objects. Finally HtmlParse class provides the API to actually parse a given page with a given option of extraction (words/images/links).

```
public class HtmlParse {
  public Vector parse(String fileName,int Type)throws IOException{
HtmlDocument document;
document = new HtmlParser(new FileInputStream(fileName)).HtmlDocument();
HtmlScrubber j = new HtmlScrubber(HtmlScrubber.DEFAULT_OPTIONS
                                  | HtmlScrubber.TRIM_SPACES);
    switch(Type){
            // words of a page
             case 3:  document.acceptWords(j);
            // images of a page
             case 2:  document.acceptImages(j);
       // links in a page
             case 1:  document.acceptLinks(j);
       }
…
}//HtmlParse
```

7.2 Implementation of Change Detection

Once a page is filtered into data of interest i.e., allWords, allLinks and allImages vectors, these vectors are used for detecting changes of interest to users. For image and links change, allImages and allLinks respectively are used for change detection. For keywords and phrase detection, they are extracted from the allWords vector. In general change detection is achieved through three phases namely,

1. Object identification and extraction
2. Change Detection
3. Change Presentation

7.2.1 Object Identification and Extraction

As mentioned in the previous section, a given HTML page is filtered into allLinks, allImages and allWords. Keywords and phrase are extracted from the allWords vector as the order of occurrence of words in the page and in the vector is same, in other words the order of occurrence is maintained during the extraction process. For detecting changes to all-words in the page; we use LCS on the allWords vector and for all other change types we use CH-Diff(as the objects of interest are extracted out from the page and hence loose the relative order maintained in the page)

ChangeDetector consists of many member functions that are implemented for detecting changes to keywords, links, images and allwords in a page. Table 7.3 shows the various member functions of ChangeDetector class along with inputs given to these functions, most of these functions are public in nature. In what follows we will discuss the implementation details for detecting changes to each supported change type.

changeDetection member function is responsible for generating the required change table, which has the following fields

change[ ][0] = word/link/image

change[ ][1] = old word count

change[ ][2] = new word count

change[ ][3] = status(Insert/Delete)

The input to this function which is two arrays iOld and iNew where the iOld array is sorted based on the hashcodes and forms a window of objects against which the elements of iNew array are compared.

```
public String[ ][ ] changeDetection(String iOld[][], String iNew[][]){


String[ ][ ]iold = wordCountTable(iOld);
 String[ ][ ]inew = wordCountTable(iNew);
```

wordCountTable returns an array of elements with their occurrence count. Initially the sorted iold array is copied into the ichange array.

```
String ichange[][] = new String[changeSize][4];
for(int i =0;i<oldSize;i++){
                ichange[i][0] = iold[i][0];
                ichange[i][1] = iold[i][1];
                ichange[i][2] = "-1"; // setting to no insert
};
```

```
for(int i =0;i<newSize;i++){
    // out of bounds [lowerbound,upperbound]
       if( (inew[i][0].hashCode() < ichange[0][0].hashCode()) ||
          (inew[i][0].hashCode() > ichange[oldSize-1][0].hashCode())) {
             ichange[ j ] [3] = "i";
             ichange[ j ] [0] = inew[i][0];
      ..

   //The new value is equal to the lowerBound
       else if(inew[i][0].hashCode() == ichange[0][0].hashCode()){
        if( newcount == oldcount) ichange[0][3] ="n";
        else if(newcount > oldcount) ichange[0][3] = "i+";
        else if(newcount < oldcount) ichange[0][3] = "d-";

//The new value is equal to the upperBound
       else if(inew[i][0].hashCode() == ichange[oldSize-1][0].hashCode()){

//with in bounds
if((inew[i][0].hashCode() > ichange[0][0].hashCode())||
     (inew[i][0].hashCode() < ichange[oldSize-1][0].hashCode())) {
    // search for occurance if found flag change according to occurance count
   // if not found flag as detele
}// changeDetection
```

The above description shows the control flow in changeDetection function; in the end after all the elements in inew set are compared against the constructed window of objects, i.e., [ichange[0][],ichange[iold.length]] all the elements in ichange with ichange[ ][2]=

“-1” are marked as deletes (ichange[][3] = “d“).

In general a change is categorized as

“i” : unique insert

“i+” : multiple occurrence and one of them is a new insert

“d” : unique delete

“d- : multiple occurrence and one of them is deleted

getInsertDeleteList scans the ichange array for inserted and deleted elements .

The function that provides the required functionally for detecting changes is the changeDetection function, that implements the CH-Diff algorithm, where a window, lowerbound and upperboud of objects by sorting them in ascending order based on their hashcodes. Java1.3 provides **hashCode**() that returns hash code hash code for a String object.

```
String temp = new String(“hello”);
int hashcode = temp.hashCode();
```

The hash code for a String object is computed as

s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

using int arithmetic, where s[i] is the ith character of the string, n is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

Table 7.3 Member Functions of ChangeDetector class that perform change detection in webvigil.Detection package

Method	**Input**	**Description**
changeDetection	String iOld[ ][ ], String iNew[ ][ ]	Implements the CH-Diff
keyWordsChangeDetector	String iOld[ ][ ], String iNew[ ][ ], String keyWords[ ]	Detects changes to keywords using changeDetector.
keyWordsGrabber	String array[ ][ ], String keyWords[ ]	Extracts keywords from the given input array.
linkChangeDetector	String iOld[ ][ ], String iNew[ ][ ]	Detects changes to links/images
allWordsDetector	String iOld[ ][ ], String iNew[ ][ ]	Detects inserts and deletes using LCS.
sort	String array[][]	Sorts the input based on hashcode.
getInsertDeleteList	String changeTable[][]	Returns a vector with inserted elements followed by deleted.

For detecting changes to keywords, the keywords need to be extracted from the allWords vector, which is a vector of words extracted from the page preserving the order of occurrence of the words. The keyWordsChangeDetector function is responsible for detecting in change to keywords, it takes old and new array of words, extracts the keywords from the given arrays and uses the changeDetector for detecting the changes.

```
iOld = p.parse("Old.html",3);    // words mode of extraction: 3
iNew = p.parse("New.html",3); // words mode of extraction: 3
ChangeDetector handle = new ChangeDetector();
change = handle.keyWordsChangeDetector(handle.sort(arrayOld),
                                        arrayNew,
                                        keyWords);
iNew =  handle.getInsertDeleteList(change);
```

KeyWordsGrabber generates an occurrence count for each unique keyword from the given input array of words. An increase or decrease in the occurrence count is taken as an insert or delete respectively.

```
public String[ ][ ] keyWordsChangeDetector(String iOld[ ][ ],
                                        String iNew[ ][ ],String keyWords[ ]){
return changeDetectoion(sort(keyWordsGrabber(iOld,keyWords)),
                        keyWordsGrabber(iNew,keyWords));
}/*keywords change detector*/
```

For links and images the allLinks and allImages vectors are used for detecting changes, again, as in keywords, an occurrence count of each unique image or link is computed and hence is used a measure for change detection.

Table 7.4 Member functions in PhraseDetector class in webvigil.Detection package

Method	**Input**	**Description**
phraseChangeDetector	String[ ] oldTextArray, String[ ] newTextArray, String[ ] phrase	Detects inserts, deletes and updates to the given phrase.
getContextBox	String[ ] textArray, int[ ] posArray, int phraseLength	Extracts the words surrounding the phrase
kmpMatcher	String[ ] textArray, String[ ] phrase	Generates a position(start position) array for all occurrences of phrase in the given array.

7.2.2 Change detection of Phrase

Unlike in keywords, links and images, for phrases we not only detect insertions and deletions, we also detect update to a given phrase. We detect an update to a phrase only if the amount of update is lower than a given limit otherwise we take it as a deletion. In order to identify a phrase we use the words surrounding it as a signature of the phrase. PhraseDetector provides the functionality for detecting changes to phrases and is a part of the webvigil.Detection package. Table 7.4 shows the Member functions in PhraseDetector class. The steps involved in phrase change detection are :

1. Identification of each instance(occurance) of phrase in old and new sets.
2. Extraction of the signature of each occurance of phrase in old and new sets
3. The extracted signatures from old and new are fed to changeDetector.
4. The elements in old marked as deleted are checked for update.

```
// testPhrase
PhraseDetector pc = new PhraseDetector(3,0.6);
pc.phraseChangeDetector(old,news,phrases);

//PhraseDetector
public class PhraseDetector {

    public PhraseDetector( int contexBoxValue, double updateFactor){
        contextBox = contexBoxValue;
        this.updateFactor = updateFactor;
  …….
 }//PhraseDetector
```

The default constructor of PhraseDetector sets the size of context box to consider and percentage of change that is considered to be an update or delete. The phraseChange-Detector uses KMP for matching the given phrase in the old and new text arrays. The getContextBox function retrieves words surrounding the phrase which make up its signature.

```
public void phraseChangeDetector(String[] oldTextArray,  String[]
newTextArray, String[] phrase){
String[ ][ ] oldContexBox = getContextBox(oldTextArray,
                vectorToArray(kmpMatcher(oldTextArray,phrase)),phrase.length);
String[ ][ ] newContexBox =  getContextBox(newTextArray,
vectorToArray(kmpMatcher(newTextArray,phrase)),phrase.length);
```

```
ChangeDetector handle = new ChangeDetector();
oldContexBox = handle.sort(oldContexBox);

String[][] changeList =  handle.changeDetector(oldContexBox,newContexBox);
Vector iNew =  handle.getInsertDeleteList(changeList);
```

Once a vector of inserted and deleted elements are obtained , for all those(signatures) that have been flagged as deleted in the old array of signatures, we extract the contents between the contextBox defined by this signature in the new array and compute LCS between the extracted data and the given phrase. If the length of the longest common sequence between these two is less than the updateFactor , a delete is flagged else it is deemed as an update.

```
for(int i=0;i<iNew.size();i++){
   // only for those entries that are flagged as delete
        if((iNew.get(i).getChange()).equals("d")){
          filterStr =getDataBetweenContexBox(newTextArray,
                                              lowerBox, upperBox);
          update = new LCSequence(phrase, filterStr)
          update.lcs();
if (update.getLCSSize()*1.0/(phrase.length*1.0) > updateFactor)
             //Updated
else
             // Deleted}
```

7.2.3 Implementation of change presentation for links, images and keywords

Currently for images and links we support change presentation by present changes in a tabular format, indicating the insertions or deletions.

```
//testLinks
change = handle.linkChangeDetector(arrayOld, arrayNew);
DisplayFunctions file = new DisplayFunctions();
file. generateTabularFormat(change,"link.html");
```

```
public void generateTabularFormat (String array[][],String fileName) throws
 IOException{
fw = new FileWriter(fileName,true);
String buffer = new String("<title>"+fileName+"</title><body>");
buffer +="<table>";
for(int i=0;i<size;i++){
      buffer +="<td "+ array[i][0]+ +"</td>";
     // depending upon the change display the change array[i][3]
   buffer += “<tr>”;
}
buffer +=”</table></body>”;
fw.write(buffer);
```

Table 7.5 shows the functions that implement the supported displaying schemes and are member functions of class DisplayFunctions.

Table 7.5 Various presenting schemes supported

Method	Description
generateTabularFormat	Generates a single frame with a table showing changes.
mergeOutputUsingFrames	Generates frames old and new, linked by a base page.

For keywords we generate a frame for each version of a page (old/new) with unique inserts and deletes marked/highlighted as show in Figure 5.14.

CHAPTER 8

CONCLUSION AND FUTURE WORK

WebVigiL is envisioned as a complete system that allows monitoring and notification of changes to structured documents in a distributed environment. WebVigiL is a system currently under development at UTA for providing an alternative paradigm for monitoring changes to the web (or any structured document). The contribution of this thesis towards the system was in the following areas:

- Design and Development of Interval and Best-Effort Based Fetching
- Best-Effort-Algorithm (BEA)
- CH-Diff: Detection of customized changes based on user intent in HTML pages.

Fetch rules to support both interval based and best-effort based fetching have been developed and tested for performance against pages that were changing at various controlled intervals on a remote testbed. Best-Effort-Algorithm (BEA), used by the best-effort rule for tuning the fetching interval to changing interval has been implemented. Various experiments were conducted to find the ideal combination of the parameters that control the adaptability of BEA. For example experiments were conducted with variable sizes of change history of a page to choose an optimal size that resulted in better performance of BEA over the others. The fetch rules have been designed to fetch a page only when there is a change to the page properties of a page or when there are no such properties available (dynamic pages). In addition various fetch rules are synchronized to prevent them from fetching the same version of a page, as their fetching process is independent from the rest.

Various change types were identified based on user intent and a detection algorithm for each change type has been developed. Currently supported change types are anychange, links, images, allwords, keywords, and phrase. A generalized change detection mechanism termed as CH-Diff was designed and implemented; this forms the core of change detection for all change types. These algorithms were tested for correctness. Finally various presentation schemes have been developed for presenting change information after a change has been detected.

8.1 Future work

Currently a page is fetched based on change in page properties. By doing so we have been restricted to handing pages with-out frames. A page with multiple frames has a base page, in which the reference to the pages in frames is given. Hence the base page is a set of references to various other pages. When a fetch rule is initiated on such a base page, by the current implementation, we base our fetching on change to page properties. We can be faced with a cases where the pages specified in the frames are changing, but the change is not reflected on to the base page, as it only contains the reference to pages and hence have no change to the page properties to base page. The only way to get around this problem is to fetch each page every time, and checks for frames and if present fetch the frames too (only on change to their page properties). Hence the fetching is done recursively to depth 2, i.e. the base page and the pages specified in frames need to be fetched.

For change types images, links and keywords we are currently detecting if a change has occurred or not and capture a change in terms of inserted or deleted. Since we extract the object of interest from the page for change computation, currently we cannot identify which instance of the object in the page has been modified. Unlike the other change types, for phrases we not only identify inserts and delete we also detect update to phrase. Here we

capture additional information for each phrase called the signature of the phrase, which is used to identify a phrase in a page. The assumption here is that the words that constitute the signature of a phrase are relatively stable. Dynamic configuration of the size of the signature along with issues where there is a change to the signature need to be resolved.

REFERENCES

[1] J.W.Hunt and M.D.Mcllroy, *An algorithm for efficient file comparison*. 1975, Bell Laboratories: Murray Hill, N.J.

[2] E.Myers, *An O(ND) difference algorithm and its variations.* Algorithmica, 1986. **1**: p. 251-266.

[3] S.Wu, U.Manber, and E.Myers, *An O(NP) sequence comparision algorithm.* Information Processing Letters, 1990. **35**: p. 317-323.

[4] Hirschberg, D., *Algorithms for the longest common subsequence problem.* Journal of the ACM, 1977: p. 664-675.

[5] Douglis, F., et al., *The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web*, in *World Wide Web*. 1998, Baltzer Science Publishers. p. 27-44.

[6] Saeyor, S. and M. Ishizuka. *WebBeholder: A Revolution in Tracking and Viewing Changes on The Web by Agent Community*. in *WebNet98*. 1998.

[7] Chen, Y.-F. and E. Koutsofios. *WebCiao: A Website Visualization and Tracking System*. in *WebNet97*. 1997.

[8] Lucca, G.D., et al. *Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages*. in *Seventh IEEE Workshop on Empirical Studies of Software Maintenance*. 2001. Florence, Italy.

[9] Baker, S.B. *A theory of parametrized pattern matching:algorithms and applications.* in *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. 1993.

[10] Balazinska, M., et al. *Advanced clone-analysis to support object-oriented system refactoring*. in *Seventh Working Conference on Reverse Engineering*. 2000.

[11] Ulam, S.M. *Some Combinatorial Problems Studied Experimentally on Computing Machines*. in *Zaremba*

S.K., Applications of Number Theory to Numerical Analysis. 1972: Academic Press.

[12] Cho, J. and H. Garcia-Molina, *Estimating Frequency of Change*. 2002.

[13] Lu, B., S.C. Hui, and Y. Zhang. *Personalized Information Monitoring Over the Web*. in *ICITA 2002*. 25-28 November 2002. BARTHUST, AUSTRALIA.

[14] Douglis, F., et al. *WebGUIDE: Querying and Navigating Changes in Web Repositories*. in *Fifth International World Wide Web Conference*. 1996. Paris, France.

[15] Nguyen, B., et al. *Monitoring XML Data on the Web*. in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 2001.

[16] Liu, L., C. Pu, and W. Tang. *WebCQ: Detecting and Delivering Information Changes on the Web*. in *Proceedings of International Conference on Information and Knowledge Management (CIKM)*. 2000. Washington D.C: ACM Press.

[17] Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.

[18] Chakravarthy, S., et al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, in *Proc. Int'l. Conf. on Very Large Data Bases VLDB*. 1994: Santiago, Chile. p. 606--617.

[19] Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611.

[20] Dasari, R., *Design and Implementation of a Local Event Detector in Java*, in *CISE*. 1999, Univ. of Florida: Gainesville.

[21] HTML-Parser.

[22] Java1.3, *http://java.sun.com/j2se/1.3/docs/api/*.

[23] Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments*. 2001, The University of Texas at Arlington.

[24] Anwar, E., L. Maugis, and S. Chakravarthy, *A New Perspective on Rule Support for Object-Oriented Databases*, in *1993 ACM SIGMOD Conf. on Management of Data*. 1993: Washington D.C. p. 99-108.

[25] Chakravarthy, S., et al., *Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules.* Information and Software Technology, 1994. **36**(9): p. 559--568.

[26]. j, B. and C. Hollander, *{DART}: An Expert System for Computer Fault Diagnosis.* 1981: p. 843--845.

BIOGRAPHICAL INFORMATION

Naveen Pandrangi was born on December 11, 1978 in Vijayawada, India. He received his Bachelor of Science degree in Information Science and Engineering from Bangalore University, Bangalore, India in May 2000. In the Spring of 2001, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington, in May 2003. His research interests include active, mobile databases and Web technologies.