APPROACHES TO IMPROVE THE PERFORMANCE OF STORAGE AND PROCESSING-SUBSYSTEMS IN WEBVIGIL

The members of the Committee approve the master's thesis of Ajay Eppili

Sharma Chakravarthy
Supervising Professor _____

Alp Aslandogan _____

Gautam Das _____

To my Family and Friends.

APPROACHES TO IMPROVE THE PERFORMANCE OF STORAGE AND PROCESSING-SUBSYSTEMS IN WEBVIGIL

by

Ajay Eppili

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2004

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his constant guidance and support, and for giving me a wonderful opportunity to work on this topic. I am grateful to Dr. Gautam Das and Dr. Alp Aslandogan for serving on my committee. I would like to thank Raman Adaikalavan, Shravan Chamakura and Akshaya Arora for helping me throughout in times of need. Thanks are due to my seniors Jyoti Jacob and Naveen Pandrangi, Anoop Sanka and Alpa Sachde for their fruitful discussions and guidance. I would like to thank my friends Ashwin Tubati, Manu Aery, Vamshi Pajjuri, Anupama Mutt, Dhawal Bhatia, Srihari, Vihang Garg, Sunith, Nikhil and my roomates for their cooperation and support.

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

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

November 19, 2004

ABSTRACT

APPROACHES TO IMPROVE THE PERFORMANCE OF STORAGE AND PROCESSING-SUBSYSTEMS IN WEBVIGIL

Publication No. _____

Ajay Eppili, M.S.

The University of Texas at Arlington, 2004

Supervising Professor: Sharma Chakravarthy

World Wide Web has gained lot of prominence with respect to retrieving information and data delivery. Users are interested in monitoring selective changes to contents than merely surfing on the web. Selective content monitoring of the web requires an effective monitoring system for change detection and notification based on user requirements. WebVigiL is a profile-based system that monitors, retrieves, and detects specific changes to HTML and XML pages on the web and notifies users in a timely manner.

The first prototype concentrated on the functionality of the WebVigiL system. This thesis investigates improvements to the performance and reliability aspects of the first prototype in a number of ways. First, a novel object reuse strategy (similar to a buffer manager) has been proposed to reduce the number of times an object is retrieved from disk and converted into an in-memory object for change detection. Second, a diff-based version manager has been implemented to reduce the disk storage growth thereby improving the scalability of the system. Third, as the WebVigiL system is prone to system failures, recovery has been added to make sure that the WebVigiL server can recover

gracefully after a failure and continue to monitor sentinels. Of course, no monitoring is possible during its downtime. However, previously defined sentinels will continue to be monitored after recovery as if the system never went down. Intelligent fetching of monitored pages is currently done at the WebVigiL mediator. As part of this thesis, a server-side (at the web site where the page resides) fetch has been introduced and its effectiveness has been analyzed with respect to the data transfer and the number of fetches as compared to the mediator-side fetch.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 World Wide Web

World Wide Web is evolving as a universal repository of shared information. Data on web is changing and increasing at an exponential rate. The explosive growth of information on the web necessitates an effective and efficient retrieval of information from the web, tracking changes and delivering it according to user requirements. Users may be searching for specific information or interested in changes to specific web pages. For example, students may be interested in updates to their course web pages. As another example, people interested in buying products on internet may want to monitor those pages for deals and hence may want to monitor occurrences of new links with specific keywords or phrases. Nowadays there are discussion groups for almost every topic that one can think of. Members in the group may be interested in new posts or updates on topics of their interest. To keep up with with changes occurring to pages push paradigm has been used traditionally.

Pull Paradigm [1] is an approach where the user performs an explicit action of querying the pages of interest on a periodic basis. Here the burden of retrieving the required information is on the user and may result in changes being missed when a large number of web sites need to be monitored. In the push paradigm [1], the system is responsible for detecting changes in the best possible (or optimal) way based on user profiles and informs/notifies the user (or a set of users) when a change occurs. At present most of the systems use a mailing list to send changes to all its subscribers. Though this approach reduces the burden on the user by avoiding continuous polling, naïve use of

a push paradigm results in informing users of the changes in a general way and may not serve specific interests of the users'. The emphasis should be on selective change monitoring and notification rather than any change. That is, notification to the user should be specific, based on user interest expressed in the form of a policy. Hence, an approach is needed to capture the users' interests, intelligently fetch the required pages, detect changes, and notify relevant changes to the users in a timely manner.

1.2 WebVigiL

WebVigiL [2, 3] is a general-purpose, information monitoring and notification system. WebVigiL uses a combination of push and intelligent pull paradigms. It uses a learning-based pull paradigm to retrieve pages of interest from remote sources, detects changes of interest and uses the push paradigm to notify detected changes. WebVigiL uses active capability to initiate a series of asynchronous operations that occur at different points in time. Event-Condition-Action (or ECA) rules are used to capture the active capability in WebVigiL. In ECA rules, on the occurrence of an event, corresponding condition is evaluated and associated actions are performed if the condition evaluates to true. In WebVigiL, ECA rules are used to handle operations for creation of a sentinel, monitoring the requested page, detecting changes of interest, notifying the users of the change, and deactivation of a sentinel.

WebVigiL architecture consists of modules to handle the specification, management, and propagation of changes as requested by a user while meeting the quality of service requirements. The modules to perform these tasks are: *user specification module*, *verification module*, *knowledge base*, *change detection module*, *fetch module*, *version management module*, *notification*, and *presentation module*. User specifies his requirements in the form of a profile/policy using the *user specification module*. Such profiles/policies are termed as *sentinels* in WebVigiL. Once a sentinel is specified, various modules of

WebVigiL carry out the required tasks. *Verification module* validates the sentinels and stores this information in a database (currently Oracle) using the *knowledge base module*. ECA rule generator of the *change detection module* is responsible for generating the ECA rules for the run time management of a validated sentinel. *Fetch module* is responsible for retrieving the required pages; these pages are managed by *version management module*. *Version management module* manages server based repository service that maintains and provides versions of pages. *Change detection module* uses this module to retrieve the appropriate pages and performs change detection between them. *Change notification* and *presentation modules* inform the user about the changes and display the detected differences between the two pages in an easy to understand format.

1.3 Motivation

The current modules of WebVigiL have been designed to achieve the desired functionality of the system. The next step is to address the reliability, scalability, and performance improvements of the system. This thesis identifies subsystems in WebVigiL that can be enhanced to improve the above and proposes approaches for the same and evaluates them. Below, we highlight the issues that form the focus of this thesis.

First, to detect a change for a URL page, two versions of a page are needed. Data from each version is extracted and organized into a runtime object. Changes are detected by comparing the in-memory objects of the two versions. Often there might be situations where more than one sentinel is interested in changes to the same versions of a URL. This may result in accessing the file from the disk (file I/O) and computing runtime objects for the same versions multiple times. A version-cache approach to maintain objects for the versions that are required more than once and reuse them when required would improve the performance of change detection by reducing the number of disk accesses and processing.

Second, the current approach of storing the retrieved pages in its entirety will consume considerable amount of disk space as the number of sentinels in the system increases. A mechanism to reduce the disk space growth as the versions in the repository increase is needed to attain scalability of the system.

Third, WebVigiL system is prone to failures. Failure of the system can occur due to system crashes, system errors and disk failures. A failure in the system will stop the active sentinels and halts monitoring changes to requested pages. The runtime information maintained by the system is lost. Failure detection and appropriate measures to recover are required for attaining reliability and stability.

Finally, currently a fetch module residing at the WebVigiL server fetches pages of interest to retrieve and store new versions of pages in the repository. Freshness of a page is determined from the metadata or page properties using the last modified time stamp or the page size. For pages without time stamp information (i.e., dynamic pages), checksum is used as a measure of freshness. When the system fetches the metadata, a connection is established to the requested web server and metadata is sent over this connection. A page is retrieved only if a change is observed in the metadata with respect to the latest page in the repository. To retrieve the page, another connection need to be established. In the case of pages where checksum is used, the page has to be retrieved in every cycle to compute the checksum. The fetching costs would be an overhead while polling for a page that changes infrequently. An approach to reduce such unnecessary processing and wasteful connections is needed.

1.4 Contributions

In this thesis the above discussed issues have been addressed. An object reuse approach has been proposed to reduce the number of disk accesses and processing while performing change detection. A version-cache has been introduced to persist the runtime

objects computed for a version and use them when required. A replacement policy that uses the information in the system to efficiently manage the cache has been shown to perform better than conventionally used replacement policies (e.g., LRU, MRU, FIFO). The approach has been integrated into the WebVigiL system.

A diff-patch approach has been incorporated into the current version controller to maintain the versions. This has been implemented to reduce the disk storage growth. Experimental results indicate a significant reduction in storage space with this approach.

Timely detection of failures and recovery from failures have been provided to achieve reliability and stability. A recovery module has been added to the current system which persists information in the runtime data structures to a stable storage using write ahead logging (or WAL). During the recovery process, these log files are used to attain a consistent state. Once recovered, all the sentinels that have to be monitored are restarted. From a user's perspective, monitoring does not happen only during the downtime.

In addition to the above, a server-side fetch mechanism to reduce wasteful connections has been proposed and its cost effectiveness is compared to the existing WebVigiL-side fetch strategy.

The outline of this thesis is as follows, chapter 2 gives an overview of the current architecture of WebVigiL and its modules. Chapter 3 discusses a version-cache approach to reuse the processed information for computing change between pages. Chapter 4 addresses the diff-patch approach for maintaining the repository of versions and chapter 5 explains the fail detection and recovery mechanisms required for the stability of WebVigiL. Chapter 6 discusses server side fetch mechanism. Chapter 7 describes the related work and finally chapter 8 outlines conclusions and future work.

CHAPTER 2

WEBVIGIL ARCHITECTURE

WebVigiL is a profile based change detection and notification system that monitors changes to structured and unstructured documents. Currently, it handles HTML and XML documents. WebVigiL architecture is modular and extensible. It has been designed to facilitate change specification, provide truly asynchronous approach to manage user requests and notify changes using active capability. Figure 2.1 summarizes the high level WebVigiL Architecture. The system comprises of various modules to specify, manage and propagate user requests to monitor web pages at different levels of granularity [4]. This chapter gives an overview of the current architecture and briefly discusses existing modules in WebVigiL [2, 3].

2.1 User Specification module

User specification module provides an interface where one can specify his/her requirements for monitoring. Users can specify their interests in terms of profiles which capture the information required for change detection and notification. Users specify the following details.

- Page to be monitored (URL).
- Type of content(specific keywords/phrases/links/images) to be monitored.
- Frequency at which the page has to be monitored (a user can specify an interval or can leave it to the WebVigiL system)
- Start and End time for the monitoring request (defaults are provided)
- The medium and frequency for notification of changes.

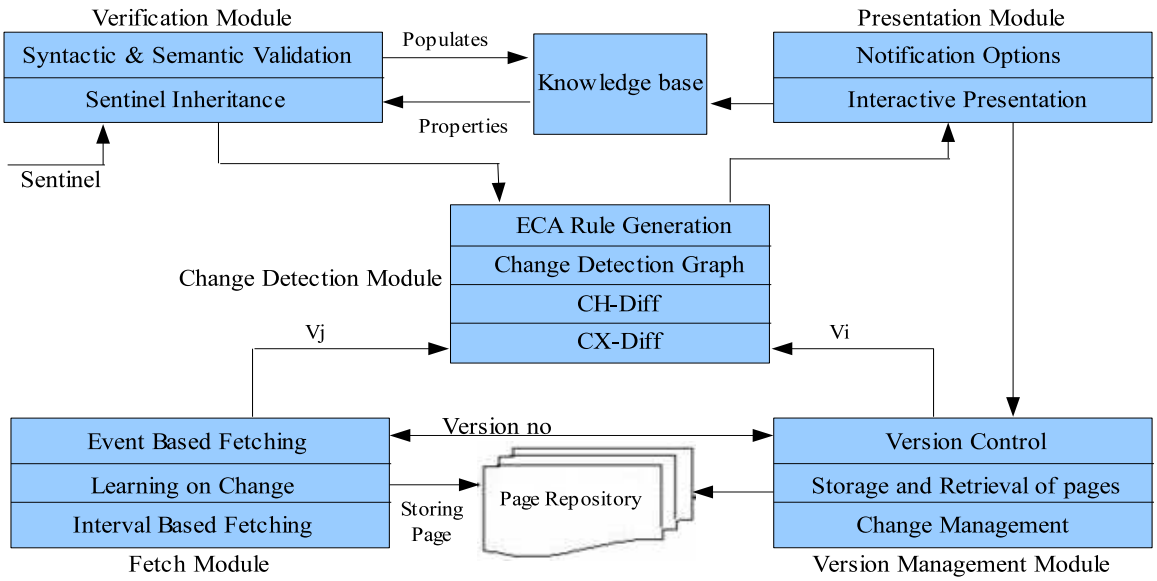


Figure 2.1 WebVigiL Architecture

- The relative version for comparison for detecting changes

WebVigiL provides an expressive language with well-defined semantics for specifying these monitoring requirements [5, 6]. Sentinel captures the monitoring requirements of a user. Different features of this specification language are summarized below:

A user can specify the type of content to be monitored in the page at different levels of granularity. A web page can contain information in the form of text, links to other pages and images. The language allows the user to request for changes to images, links, specific keywords or phrases. Users can even place a request with the above mentioned change types or a combination of them (for example changes to links and images) using AND, OR, and NOT operators. The language also allows the users to set sentinels on their own previously defined sentinels which provides a way to keep track of correlated changes. Unless the user explicitly specifies the properties, the new sentinel inherits all the properties of the previous sentinel. The language provides a way of specifying the fetch frequency which is the frequency by which users want the system to check for

changes in the page of interest. The options are either a user-defined frequency (e.g., every 2 days) or system-defined frequency where the system tunes the fetch frequency to the actual change frequency using a learning algorithm based on history. The medium of notifications that can be specified by the language are email, fax, PDA and dashboard. Currently email and dashboard are supported. The frequency of notification can be given as user-specified frequency or system-defined frequency. The options for the relative versions of a page to be compared are: moving, every, and pairwise. A sentinel with option moving(n) compares every fetched version of a page with the last n^{th} version in a sliding window. In every(n) every n versions of a page are compared and in pairwise the last two version of the page are compared.

For example consider the Scenario: Smith wants to keep track of flight deals to Las Vegas by monitoring changes to a travel website "http://www.hotwire.com" and desires to be notified daily by e-mail starting from November 8, 2004 to December 8, 2004. he can specify a sentinel as follows:

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

Monitor all links AND keywords "Vegas deals"

Fetch every 2 days

From 11/08/04 To 12/08/04

Notify By email smith@msn.com Every 4 days

Compare pairwise

A web-based user interface is provided to the users to specify these requirements and submit their profiles to WebVigiL system.

2.2 Verification module

Once a user provides the requirements in the form of a sentinel, it is validated on the client side. The *sentinel* is validated for its URL, start and end times of monitoring. if a

page with the specified URL does not exist or cannot be accessed a message is prompted to the user to check the URL or request for a different URL. The chronological ordering of start time and end time are validated and appropriate messages are prompted to the user. if the start of a sentinel s1 was specified as the end of another sentinel s2, and at the time of specification if s2 had already expired then a message is given to the user to change the start time. Finally the sentinel name is validated for duplication for the same user so that every sentinel installed by the same user has a unique name. Validation is completed on the client side, once the sentinel is validated it is persisted using knowledge base.

2.3 Knowledge base

The details of the validated sentinel are stored and persisted in a repository which is known as the knowledge base. Knowledge base consists of relational tables in a database. Currently, the database used is Oracle. The database is updated with important run time parameters by several modules of the system. The change detection module updates the respective relations with the status of the sentinel and recently detected changes for notifying the users. When the WebVigiL system crashes, the information from knowledge base can be used to restart the sentinels that were alive during the time of crash. The sentinels that need to be restarted can be identified based on the start time and end time of sentinels. For each sentinel that needs to be restarted, the information required by various modules for monitoring and notification is obtained from knowledge base. For instance the change detection module detects changes based on the information provided in the sentinel, such as the URL to be monitored, the change and compare specifications. 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. Queries are posed to retrieve the required data from the tables at run time with the help of a JDBC bridge.

2.4 Change Detection module

As the web is burgeoning at an astounding rate, there is a paradigm shift on how relevant information should be retrieved from the web. Users now look for selective changes on the web. WebVigiL system provides that through the change detection module.

2.4.1 ECA Rule Generator

In WebVigiL, each valid request that arrives induces a series of operations that occur at different points of time [7, 8]. Some of the operations are: creation of a sentinel (based on start time), monitoring the requested page, detecting changes of interest, notifying the user(s) of the change, and deactivation of sentinel. In WebVigiL, for every sentinel, the ECA rule generation module generates event-condition-action (ECA) rules [9, 10] to perform some of the above mentioned operations. Briefly, an event-condition-action rule has three components: an event (occurrence of an event), a condition (checked when the associated event occurs), and an action (operations to be carried out when the condition evaluates to true). The ECA rules [11, 12] along with the local event detector (LED) [7, 8] are used:

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

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

default. A sentinel can be disabled (does not detect changes during that period) or enabled (detects changes) by the user during its lifespan. In WebVigiL, the ECA rule generation module creates appropriate events and rules to enable/disable sentinels. As WebVigiL also supports sentinels defined on previous sentinels, ECA rules are an elegant mechanism for supporting asynchronous executions based on events.

Generation of Fetch Events: Periodic events [9] are defined and rules are associated for fetching with the periodicity based on the frequency of fetch specified by the user. Whenever a periodic event occurs, the corresponding rule is fired, which then checks (condition part of the rule) for a change in the metadata of the page and fetches the page (action part of the rule) if there is a change in metadata. Thus a periodic event controls both the polling interval and the lifespan of the fetch process. A fetch rule is created and used to poll the page of interest specified in the sentinel.

Generation of Notification Events: There are several options for delivering the notifications. One of the options is interval-based notification [13] in which the user specifies the frequency desired for notification of changes. Again, we use ECA rules to perform the above. The system registers with the LED and creates periodic events [9] for each sentinel that requests interval-based notification. The event fires at regular intervals corresponding to the frequency specified by the user. When a periodic event occurs, the condition part of the rule is checked for any change detected since the last notification and notifies the corresponding user (action part of the rule) if a change has occurred. A notify rule is created and used to send time-based notifications to users.

2.4.2 Change Detection Graph:

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

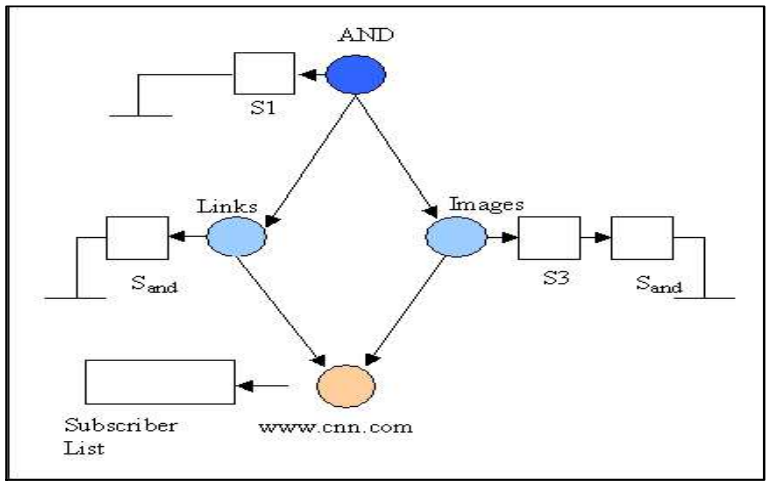


Figure 2.2 Change Detection Graph

page. The relationship between the sentinels and page is captured using a graph [14]. The graph aids the change detection module in being less computationally expensive by maintaining certain runtime information. For example, if a sentinel s1 requests on the URL "www.cnn.com" and is interested in the change type "Links AND Images", and if another user places a sentinel s3, on the same URL for "Images". The graph constructed for s1 and s3 is as shown in Figure 2.2.

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

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

greater than level-1) in the graph belong to this type. Current WebVigiL system supports composite changes on a single page.

As seen in the graph, the relationship between nodes at each level is captured using subscription/notification mechanism. All the higher level nodes subscribe to the lower level nodes. The lower level nodes maintain the subscription information in their subscriber's list of each node. The list corresponding to URL node consists of all the change type nodes that have subscribed to it. At the change type nodes each sentinel will have a subscriber that will contain the references to the composite nodes. Thus when a page is fetched, the corresponding URL node is notified and the information is propagated to the higher level nodes. The change is computed at the change type node between the most recently fetched page and the reference page that is selected according to the compare option specified for that sentinel (as discussed earlier). If the page happens to have changed the sentinels interested in the change (the sentinels in the subscriber list) are notified. The change type node being a part of the composite node, the later is also notified of the change. In the above graph change in images is computed only once for s1 and s3. As s3 is interested only in images, it is notified as soon as the change occurs. But s1 is interested in a composite change (images and links), so a change is propagated to the composite node (AND) by the change type nodes "images" and "links", only then s1 is notified.

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

Currently, change detection has been addressed for Hypertext Markup Language (HTML) and eXtensible Markup Language (XML) documents. Change detection algorithms [4, 5] work as per the change type specified by the users while giving the input. When a change has to be detected between two pages, the required change types are extracted from each page and stored in runtime objects. These runtime objects are com-

pared for changes. The changes detected are reported to the notification module for the corresponding user to be notified. In HTML, changes are detected to content-based tags such as links and images, presentation tags and changes to specific content such as keywords, phrases etc. As XML consists of tags that define the content, currently, we support change detection only to the content.

2.5 Fetch module

In order to monitor a page targeted by the sentinel, it has to be fetched using the specified periodicity. The fetch module fetches the pages of interest to the users based on the specified frequency. The options for the frequency are user-defined frequency (for e.g., every 2 days) or can be system-defined. This module informs the version controller of every version it fetches, stores it in the page repository and notifies the CDG of a successful fetch. The properties of a page are checked for freshness in a page. The properties that a page has are: Last Modified Date (LMT) or size of the page (Checksum). Depending upon the nature (static/dynamic) of page being monitored the complete set or subset of the metadata is used to evaluate the change. Only if the properties have changed, the rule fetches the page and sends it to the version controller for storage. Fetch rules are also created and used to retrieve the page of interest specified in the sentinel with the frequency specified as the interval of polling.

Best Effort (BE) Rule: In situations where the user has no information about the change frequency of a page or relegates that task to WebVigiL, it is necessary to tune the fetch frequency to the actual change frequency of a page. BE Rule uses a best-effort Algorithm (BEA) [15] to achieve this tuning. In the best-effort algorithm, the next fetch interval (Pnext) is determined from the history of changes to that page. When the next polling interval is determined, the BE Rule changes the interval "t" of the periodic event.

Interval-Based (IB) Rule: The user explicitly provides a fetch frequency. A periodic event [9, 10] with periodicity (interval t) equal to the given interval is created and an IB rule is associated with it to fetch the page. As a result, there will be more than one IB rule on a given page with different or same periodicity, where each rule is associated with a unique periodic event (i.e., with different start and end times).

2.6 Version Management module

Version Management module provides a repository service to archive and manage different versions of web pages. To compute the change between the versions of the page, the system needs to get the content by connecting to the web page. There can be situations where more than one user is interested in the same web page. In such cases, the system has to connect to web page multiple times for the same version of the page. Also, the content of the versions will be required for the presentation module. The repository service is to reduce such network connections by storing the pages and using them when needed, thereby reducing network traffic. But as the versions are fetched for every URL in the system, there might be a storage overhead on the system. To avoid this only those versions needed by the system are maintained in the system. Version controller module triggers a deletion process to purge all the versions that are not required for any sentinels [13]. The deletion algorithm considers the specifications of the sentinels requesting a particular web page to identify the set of versions that are required for the system. It deletes all the older versions.

2.7 Presentation & Notification module

The primary functionality of this module is to notify the changes to users in multiple ways including presentation in an understandable format. A user has to be given a quick

idea of what has been changed. Two schemes are used to present the changes to users, viz., only-change approach and the dual frame approach. *Only-change approach* is designed to give a brief overview of the changes, the changes in terms of links, keywords, images added/deleted to page are summarized in a tabular format. *Dual-frame approach*, in contrast, has the old and new versions of the page shown side by side, with the changes highlighted. As the HTML tags are used for presentation and XML tags define content, the presentation of HTML and XML differ [13].

Users have a choice in the medium of notification and the frequency by which they want to get notified of the detected changes. The notification module delivers notifications to the users with a frequency that was opted during the installation of the sentinel. Currently the medium of notification is email only. The functionality of WebVigiL also provides the users with a dashboard where the users can visit the WebVigiL dashboard and look up their changes and manage their sentinels. The WebVigiL server, based on the notification frequency can push the information to the user, thus implementing the "just in time"(JIT) paradigm.

Current WebVigiL system does not start as a separate application. The modules of WebVigiL are started only when a request for monitoring is placed. Using Java Server Pages (JSP), the interface for specifying a sentinel is provided for users. This interface runs on the servlet engine. When a sentinel is submitted for the *first time* verification module validates it and informs the user in case there are any invalid specifications. If the sentinel is valid, the modules required for WebVigiL are initialized and using the *knowledge base module* the details of sentinel are persisted. The sentinel is then sent to change detection module to generate ECA rules according to its specifications. The modules are initialized in the address space only for the first time a sentinel is placed. For subsequent sentinels, the modules already initialized are used to carry out the required tasks.

CHAPTER 3

VERSION-CACHING FOR CHANGE DETECTION

Change detection module detects changes to HTML and XML pages. Separate algorithms have been developed [5, 16] to compute changes for HTML and XML pages, as they differ in the way tags are interpreted. Change detection module detects changes by between two versions of a page by applying appropriate change detection algorithms based on their page type (HTML/XML). Change detection algorithms (for both HTML and XML) perform pre-processing steps to create runtime objects of the versions involved. The pre-processing steps for a version involve accessing the file from the disk, extracting the contents from page(HTML/XML) and organizing the content into a runtime object. Changes between two versions are detected by comparing the runtime objects created for the two versions. Once the changes are detected, they are notified to users based on their interests, and the runtime objects are deleted.

In WebVigiL, there can be situations where same versions of a page are required more than once for change detection. If the same version is required multiple times, similar steps have to be repeated to create the runtime object. Given the costs involved in disk access and page parsing, it becomes imperative to reduce the costs whenever possible. This can be achieved by storing and reusing the runtime objects of versions that are required more than once. A version-caching mechanism to maintain the runtime objects of different versions and reuse them when needed is presented in this chapter. This chapter also discusses the pre-processing steps required for HTML and XML documents and gives an overview of the objects being reused.

3.1 Version-caching mechanism

The reason for introducing a version-caching mechanism in WebVigiL is to store the runtime objects in an in-memory cache and reuse them when needed. When a version is needed by the change detection module for the first time, the runtime objects are created using the appropriate algorithm (HTML/XML), and stored in the cache. If the same version is needed again by the change detection module, the runtime object of the version is obtained directly from the cache. The cache design should facilitate the management of the stored objects and be able to use a pre-specified number of objects (similar to a buffer manager or cache). It should retain objects which will reduce the number of disk accesses and processing and remove objects that will not be required. It should maintain the required information with the objects in order to perform such operations. The following section discusses the factors that are relevant for maintaining runtime objects in the version-cache.

3.1.1 Maintaining runtime objects

The cache should maintain only the runtime objects of versions that can be reused. When there are more number of runtime objects that need to be cached, strategies have to be applied to maintain the objects that have better reusability. This calls for a way to handle the objects in the version-cache and a replacement policy to choose an object to be replaced when the cache overflows (or does not have space for an incoming object).

In operating systems and DBMSs, information of the pages that will be required in future cannot be determined. The cache mechanisms in such systems use replacement policies which determine pages to be replaced on the basis of recency (as in the case of Most Recently Used or Least Recently Used) or order of arrival (First In First Out) or utility in the past (Least Frequently Used, Most Frequently Used). LRU might not always work as the least recently used version may be reused more number of times.

In WebVigiL, as the most recent version of a sentinel is likely to be used when a new version is fetched, MRU policy might not be applicable in most of the cases. LFU and MFU policies consider the utility of the page in history. In WebVigiL, the reusability of a version may be more appropriate in maintaining its runtime object. There may be situations where these policies are sufficient to maintain the objects. But to support different scenarios, a logical approach is required which would consider the reusability of a version in the system.

The cache should store runtime objects of versions that would be required more number of times. In WebVigiL, predicting the exact access-pattern of a version might not be possible, but estimates can be made according to the requirement of the runtime object for a single sentinel. These estimates can be used to determine whether an object needs to be kept in the cache or not. A strategy to estimate the requirement of a version has been introduced. This strategy requires maintaining two factors for a runtime object. The primary factor for storing a runtime object in the cache is *utility factor*. *Utility factor* determines the number of times the runtime object of a version will be reused.

When a runtime object of a version is computed for the first time, the utility factor is estimated. The runtime object is stored in the cache only if the utility factor is greater than 0. If the cache is full, then the new object is stored only if it has a better utility than all the objects that are already in version-cache. This can be performed by first selecting a runtime object with the lowest utility factor from the cache and comparing its utility with the new runtime object to be stored. But there can be situations where more than one object in the cache has the same utility factor. There needs to be another criterion to determine a victim object among the runtime objects with the same utility factor. To handle such cases, a secondary factor has been introduced. This factor checks for the relative utility of the objects with the same utility factor. To understand the significance of frequency factor, consider the following example.

Consider a scenario where there are three sentinels S1, S2, and S3 on three different web pages,

1. S1 requests for a URL X with a fetch frequency of one hour for a week
2. S2 requests for a URL Y with a fetch frequency of one hour for a week
3. S3 requests for a URL Z with a fetch frequency of one day for a week

From the specifications, it can be estimated that over a week, relative to S1 and S2, the number of versions that are fetched for S3 are less and the number of change detections that take place for S3 are also lesser than S1 and S2. If a runtime object of sentinel S3 is in the cache, it will be used only when the next version for S3 is fetched, that is after one day. But in the case of S1 and S2, the runtime object in the cache will be used in the very next hour. If a situation arises to choose a victim object between runtime objects of S1, S2 and S3, it would be more logical to choose an object of S3, as its relative utility over a period of time is less. The objects of Sentinel S3 are required less frequently than the objects of sentinel S1 and S2. The relative utility of the objects with the same utility factor is determined from the fetch frequency or fetch interval of the version. This factor will be referred as *frequency factor*. The factors considered can be summarized as follows:

- *Utility factor* is the primary factor considered to decide the usefulness of a version in the system
- *Frequency factor* is the secondary factor which is considered for objects having the same utility factor

When there are objects with same utility factor and frequency factor, it means the effect of removing any object is the same. So, the object which appears first is chosen as the victim object. These factors have to be maintained along with the runtime objects to manage the objects in cache. The following two sections discuss the handling of the utility and frequency factor respectively.

3.1.2 Utility factor

Utility factor of a version is the number of times the version will be reused. As the exact requirement of a version cannot be predicted during runtime, an estimation approach is used here. This section discusses the difficulties involved in exactly predicting the utility factor of a version.

Predicting the exact utility of a version: WebVigiL system allows users to place sentinels with different specifications. Same URL can be requested by different users with different or same specifications of fetch rule, fetch frequency and compare options. During runtime, determining the exact utility factor of a version V_x would mean to first identify all the sentinels interested in the same URL and further exactly determine the number of times each sentinel requires the version V_x. Version manager maintains runtime data structures to store information of the versions required by every sentinel that has fixed interval as fetch type and for every group (sentinels interested on the same URL) that has best effort as fetch type. At any given time point, scanning these data structures will give the number of sentinels that are interested in the same URL but whether the sentinel requires the same version or not cannot be determined. An alternate estimation approach is discussed below.

Estimating the Utility Factor: An alternate approach is to estimate the utility factor of the runtime object of a version with respect to a single sentinel. For example consider a case where a user places a sentinel S1 with the following requirements.

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

Monitor all links

Fetch every 1 hour

From: Now To: Now + 4 hours

Compare pairwise

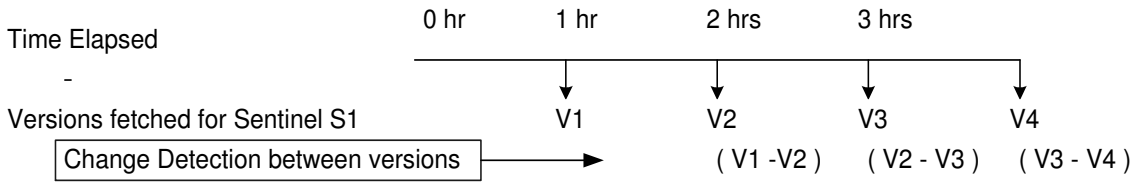


Figure 3.1 Usage of versions

When the first version is fetched, it is stored in the repository. When the second version is fetched after one hour, it is stored as V2 and change detection is triggered between versions V1 and V2. Here, V2 is the latest version for change detection and V1 is the previously fetched version. The runtime objects of V1 and V2 are computed and compared for changes. It can be estimated that as V2 is the latest version, the runtime object of V2 is required for change detection when the next version is fetched. V1 is not required for sentinel S1 once it is used for computing change with V2. Similarly, when the third version is fetched a change is triggered between the latest version for change detection (V3) and the previous version for change detection (V2). It can be estimated that the runtime object of V3 will be used when the next version is fetched and runtime object of V2 is not required for sentinel S1 any longer. Figure 3.1 summarizes the usage of each version for S1 over 4 hours

In the Figure 3.1, except for the first version and the last version, the runtime objects for all the other versions are computed twice. In such cases, the runtime objects of the intermediate versions can be saved and reused when needed. Saving the runtime object of *first version* will not be useful in this case as there is only one sentinel on the URL and it does not require first version more than once. But if there are multiple sentinels having the same version as their first version it might be useful to store the run object of the first version. To support such cases, when the first version is fetched for a sentinel, the runtime object can be computed and stored in the cache. The first

version is sure to be used for change detection. For all other versions, it may depend on the compare option. So the runtime objects of all other versions of a URL can be stored when the change detection module requires them.

The following points summarize the estimation of utility of a version *for a sentinel*.

1. For the first version fetched, the runtime object is stored in the cache with a utility factor of 1. If the object already exists, the utility factor is incremented by 1.
2. Two versions are involved in change detection: a *latest version for change detection* (V_L) and *previously fetched version* (V_P)
3. The runtime objects created for the V_L can be stored in the cache with a utility factor of 1 since that object will be used once again. If the object already exists, the utility factor is incremented by 1
4. For V_P, if it is available in the cache, the utility factor of the object is reduced by 1 to reflect that the object has been used once. If the object does not exist, then the computed runtime object need not be saved in cache as it is not required by the same sentinel at a later point of time

Utility factor of a runtime object decides whether an object needs to be in the cache or not. If the runtime object's utility factor is 0, it means that the object will not be used in future. When the cache is full, such objects are chosen for replacements as they have the lowest utility factor. This way, whenever an object is not needed, it has more chance of getting replaced.

Consider a situation where there are two sentinels S1 and S2. Let the size of the cache be 2 and let the page change after every one hour.

Sentinel S1 with the same specifications as above on a URL *www.uta.edu* with a fetch interval of 1 hour to monitor links for a period of 4 hours with a compare option *pairwise*.

Sentinel S2 with the following specifications

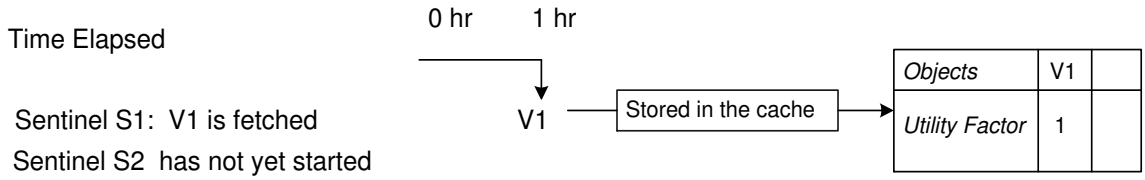


Figure 3.2 After 1 hour

Sentinel S2 on http://www.uta.edu

Monitor all links

Fetch every 2 hour

From: Now +1 hour To: Now + 4 hours

Compare pairwise

Here, the two sentinels are interested on the same URL but have different specifications. Sentinel S2 will start one hour later than sentinel S1 and will fetch pages every two hours. This implies that there might be versions fetched by sentinel S1 that may be useful to S2. Following the steps mentioned above, the versions can be stored in the cache and reused when required.

After a period of one hour: As shown in the Figure 3.2, after one hour a version is fetched for Sentinel S1. Change detection is not triggered as only one page is fetched. But as the version V1 is the first version, the runtime object is computed and stored in the cache with utility factor 1. The version-cache here is shown for understanding the basic procedure. Design and implementation of cache is discussed later in the chapter.

After a period of two hours: After two hours, a version is fetched by sentinel S1, say V2 and the same version is used by sentinel S2. V2 is the second version for S1 and first version for S2.

For Sentinel S1: The following steps are carried out

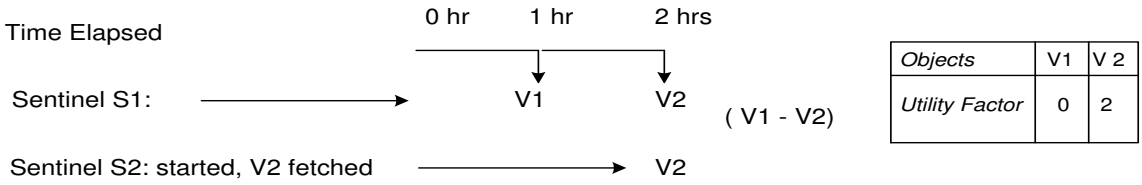


Figure 3.3 After 2 hours

1. Change detection is triggered between V2 (which is the current V_L) and V1 (which is V_P) in the case of S1.
2. During change detection, the cache is checked for runtime object of V1. As this object exists, it is used from the cache and its utility factor is reduced.
3. Version-cache is again checked for V2, but as this has been encountered for the first time, the runtime object is created. Since the object of V2 will be used by S1 when the next version is fetched it is stored in the cache.

For Sentinel S2:

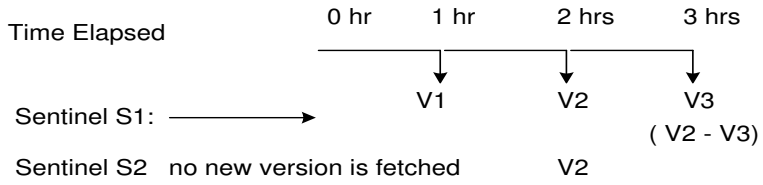
1. V2 is the first version for Sentinel S2 so it has to be stored in the cache. But as the runtime object already exists, utility factor of the object is incremented.

Figure 3.3 shows the versions fetched and the state of the cache after two hours

After a period of three hours: After three hours, a version is fetched for sentinel S1 but not for S2 as per the fetch intervals of these sentinels.

For Sentinel S1: The following steps are carried out in this order

1. V3 is fetched by Sentinel S1. A change is triggered between V3 (which is the current V_L) and V2 (which is V_P).
2. Version-cache is checked for runtime object of V2. As the object is available here, it is used and the utility factor is reduced by 1 (as it is used once). Version-cache is then checked for runtime object of V3. As the cache does not have the object of V3, it is created and stored in the cache with a utility factor of 1.



Objects	V3	V 2
Utility Factor	1	1

Figure 3.4 After 3 hours

For Sentinel S2: No version is fetched at this point

Figure 3.4 shows the state after three hours. The version V2 is still required once (by S2) and the version V3 is required once by S1.

After a period of four hours: After four hours a version is fetched by S1, say V4.

For Sentinel S1:

1. Change detection is triggered between V4 (which is the current V_L) and V3 (which is V_P).
2. Version-cache is checked for runtime object of V3. As the object is available here, it is used and the utility factor is reduced by 1. As shown in the Figure 3.5 the utility factor drops to zero.
3. Version-cache is then checked for runtime object of V4. As this object is not found, it is computed and used. It is stored in the cache with utility factor of 1. The object of V4 will be inserted in the slot which has the object with the least utility factor. The slot in which V3 existed previously is chosen here as object of V3 has utility factor of 0.

For Sentinel S2:

1. If the same version V4 is used by S2, it triggers a change detection between V4 (which is the current V_L) and V2 (which is V_P).

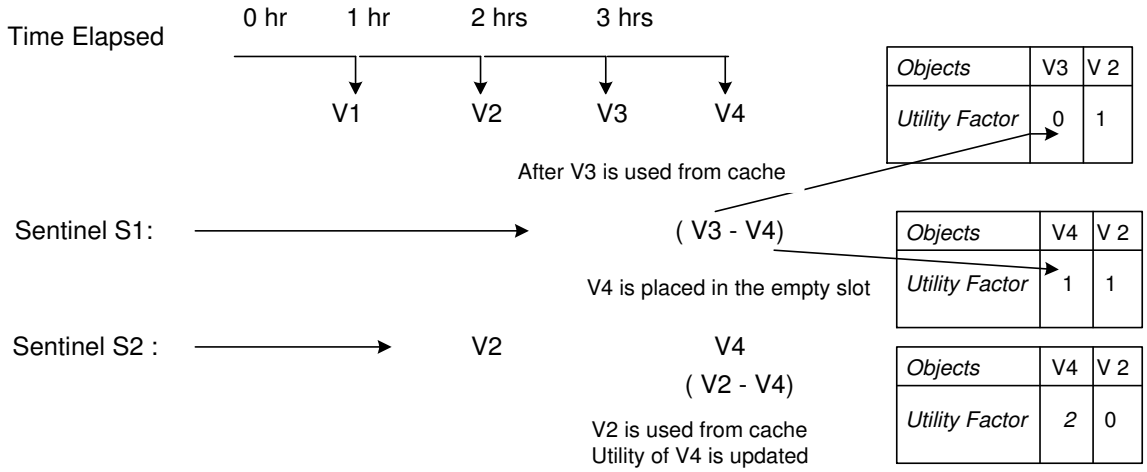


Figure 3.5 After 4 hours

2. Version-cache is checked for runtime object of V2. It is used and the utility factor is reduced by 1. The utility factor drops to zero.
3. Version-cache is then checked for runtime object of V4. It is used and the utility factor is incremented by 1 (as it is required by S2 in future).

Figure 3.5 shows the status of the cache after the change detection between V2 and V4 takes place.

It can be observed that following this approach of estimation of a version object based on the requirement of a sentinel, the cache can be used effectively to maintain objects required. The example above illustrates the case where the sentinel has a compare option of pairwise and is a fixed interval sentinel. If the sentinel has a compare option of every 5, then change is computed after every 5 versions. If first version fetched for the sentinel is V1, changes detection is triggered only when the fifth version V5 is fetched. Though the versions V2, V3, V4 are fetched, change detection is not triggered. During change detection between V1 and V5 as V5 is the latest version for change detection, runtime object of V5 is stored in the cache. The object can be used from cache when change detection is triggeres after five versions (between V5 and V10). The versions

V2,V3 and V4 will not involve in change detection so they are never stored in the cache. As the estimation is according to the versions which are required for change detection for a sentinel(that will be reused), it will work for all the types of compare options.

Last versions of a sentinel: For the above case as shown in figure 3.5, sentinels S1 and S2 end after four hours. While saving the version V4 into the version-cache, the utility factor was incremented by 1 assuming that it will be used when the next version is fetched. But as the sentinels end, the version V4 will not be used and the utility factor never goes below 2. The object will be retained in the cache as it has higher utility factor even when it is not required by any sentinel. Such versions are handled by performing the following steps.

- When a sentinel ends, the last version fetched for the version is obtained using the data structures of version controller.
- Version-cache is checked for the runtime object of this version, if it exists the utility factor is reduced by 1 (the object cannot be removed from cache as this object may be required for some other sentinels). If the utility factor reduces to 0, the object will be anyway removed from the cache.

3.1.3 Frequency factor

Frequency factor is a secondary factor used to select a victim object to be replaced when the cache is full. The purpose of this is to find the relative utility of the objects with the same utility factor. For frequency factor, the fetch frequency (in the units of minutes) of the version is considered. Versions can be required for sentinels which have fetch type as best effort or fixed interval. For versions fetched by fixed interval sentinels , the fetch interval in terms of minutes is considered as frequency factor. For versions fetched by best effort sentinels fetch interval keeps changing, so the frequency factor is not maintained for the runtime objects in version-cache. When the runtime object is

created for such a version, frequency factor is given a *null value*. Runtime objects with frequency factor as *null* are considered to have a higher frequency factor than runtime objects with a valid frequency factor.

If a version is required by fixed interval sentinel and best effort sentinel, the fetch interval of the fixed interval sentinel is considered as the frequency factor of runtime object. There can also be more than one fixed interval sentinels (with different fetch intervals) requesting for the same URL. For runtime objects of such versions, the fetch interval of the sentinel which is having the least fetch frequency among the sentinels is maintained as the frequency factor.

3.2 Runtime objects in version-cache

Runtime object of a version is required for change detection. This section discusses briefly the pre-processing steps involved in generating the runtime objects for version of HTML and XML pages and lists the objects that can be reused for a version.

3.2.1 HTML objects

CH-Diff [4] is the algorithm used in WebVigiL to detect the changes between two HTML pages. A HTML document can be viewed as a document containing raw text along with formatting and presentation markups and certain content-defining markups. Users may wish to track changes for the whole page or specifically for some elements like links, images, phrases and keywords. Changes to elements in a web page like links, images and text (in terms of keywords, phrases) can be detected using CH-Diff. When CH-Diff has to detect changes between two versions, it organizes the contents of each version into objects and compares them. Each version undergoes the following two phases:

1. *Element identification and extraction*: Elements of interests are extracted from the page. If change has to be detected for links, the links in the page are extracted. In

general, if 't' is the element of interest, then set T is defined as a set of all elements of type t extracted from the page. Implementation-wise this extraction of contents involves loading the page into memory and reading the whole file for extracting links.

2. *Organizing the data extracted into objects*: The elements in the set T are sorted and organized into a runtime object. More specifically the runtime object is a multi dimensional array containing each element along with some information. This involves sorting the contents, creating a multidimensional array and arranging the sorted elements in it.

After computing the runtime objects of each version, the change detection module applies its algorithm over these objects to detect changes. These two phases are very specific to a file and the change type (elements like links, images etc) and can be reused if the same page is needed for the same change type. In the system, as the same URL can be requested for different change types, to achieve a better reusability the runtime objects for each change type can be computed and stored in the cache. This means whenever an object has to be saved into the version-cache, the runtime objects for all change types have to be computed and saved. With such an approach the version-cache can provide objects for sentinels interested in same URL and different change types.

3.2.2 XML objects

CX-Diff [5] is the algorithm used to detect changes between two versions of a XML page. In CX-Diff, each version is considered as an ordered labeled XML tree. CX-Diff detects changes pertaining to keywords and phrases. Based on the users interest, elements are extracted from the contents of the XML document and the structural information is derived by computing the signature. Signature is computed for each extracted leaf node.

When change has to be detected between two versions each version has to undergo the following two phases

1. *Transforming into DOM Tree*: Each file is loaded into memory and transformed into a Document Object Model (DOM) object
2. *Computing Signature*: The value of the leaf nodes is extracted and their signature is computed from the element information. In this step the signature of the nodes are computed based on the keywords or phrases used by the user.

In CX-Diff as the computation of signature depends upon the keywords specified by the user. The common object that can be reused is the DOM object [17]. So when a version is required for change detection for the first time, the DOM object is saved into the version-cache.

3.3 Design aspects of version-cache

The data structures required for version-cache and their implementation details are discussed below. Figure 3.6 shows the data structures used for version-caching.

Runtime-Object is a data structure to store the runtime objects of a version. It maintains, the name of the runtime object and information of the type of page (HTML/XML) in the form of enumeration object along with the runtime objects of a version page.

The runtime objects are maintained in a hashtable with the key being the change type and the value being the actual runtime object. For a HTML page, as shown in the figure 3.6 the hashtable will have runtime objects of links, images and keywords are stored in a hashtable. For XML pages, as shown in the figure 3.6 the hashtable stores the DOM object for change type *keywords* and the object (a multi dimensional array) for change type *all words*.

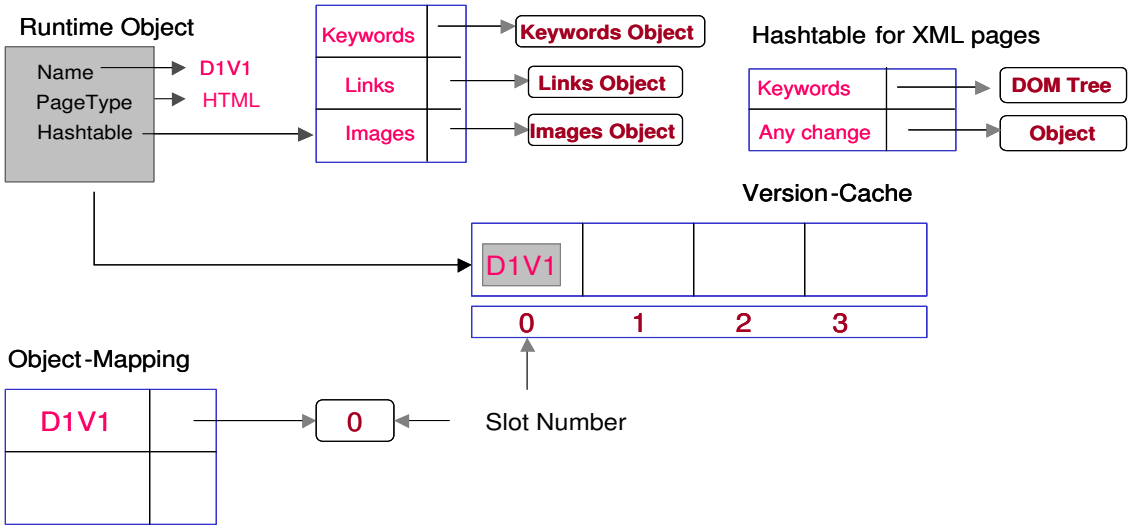


Figure 3.6 Data structures for version-caching

This design of Runtime-object helps in extendibility, in case WebVigiL supports any other change type in future the data structure of Runtime-object need not be modified. The runtime object of that new change type can be stored in the hashtable corresponding to the name of the change type.

Version-Cache is a data structure to store the Runtime-objects of different versions. Version-Cache is a like a buffer pool which maintains the Runtime-objects in the specified slot. The size of the Version-Cache is taken from the configuration files and is initialized when WebVigiL system starts. It is implemented as an array of Runtime-objects.

Object-Mapping is a data structure to provide easy access to Runtime-objects of a version. It stores the slot number of each Runtime-object in the Version-Cache. This is implemented as a hashtable with the key being the name of runtime object and the value being the slot number. As shown in the figure 3.6, if versions of a URL are stored in the directory D1 and the version that needs to be stored in the cache is V1, the name of

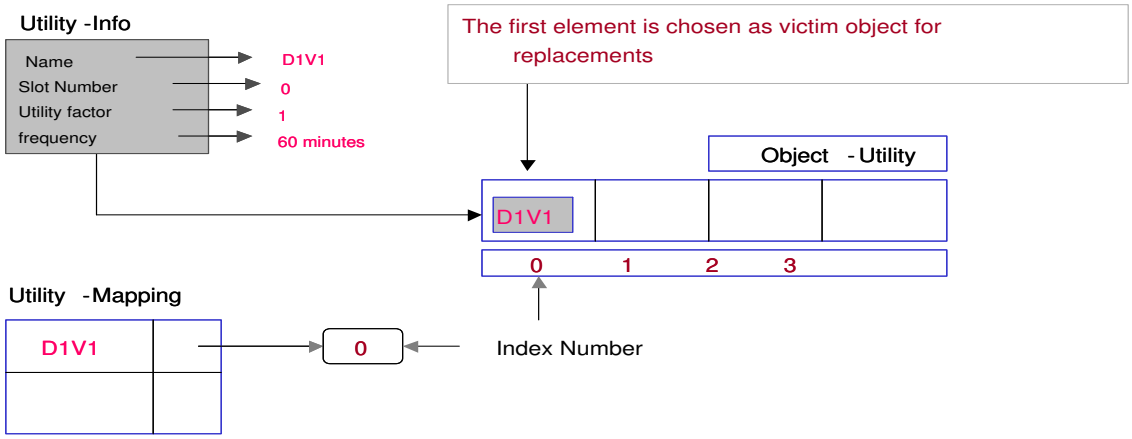


Figure 3.7 Data structures used for utility policy

runtime object is D1V1. The name is obtained by concatenating the directory mapping and version name.

The above data structures are used to access the runtime objects of versions. When the version-cache is full, the replacement policy needs to choose a victim object. As discussed in section 3.1.1, a policy which considers the utility factor and frequency factor of a version is used here. This policy is referred as *Utility policy*. Utility policy needs some information to be maintained about the runtime objects in the version-cache. This information should be updated as and when the runtime objects are created and used. The data structures used to maintain this information are discussed below.

Figure 3.7 shows the data structures maintained for utility policy.

Utility-Info is for storing the information of a runtime object such as utility factor, frequency factor and current slot number along with the runtime object name. This is implemented as a java class which stores the above parameters.

Object-Utility is a data structure to maintain objects of Utility-Info. For every runtime object in Version-Cache, Object-Utility stores a Utility-Info object. Size of the Object-Utility is same as the size of Version-Cache. Whenever the runtime object in

Version-Cache is accessed, the corresponding Utility-Info object is accessed from Object-Utility and the value of utility factor is updated. The Utility-Info objects in Object-Utility are arranged in such a way that the first element is always the least used object in version-cache. Object-Utility is implemented as a heap of Utility-Info objects. On updates to utility factor of Utility-Info objects are heapified to ensure that the first element in the heap has the lowest utility among the objects in version-cache.

Utility-Mapping: Since these objects are accessed on the basis of runtime object's name, it is important to maintain the position of a particular Utility-Info object in Object-Utility. This mapping is stored in a data structure *Utility-Mapping*. This is implemented as a hashtable, which stores the position of Utility-Info object corresponding to the runtime object name. The hashtable has name of runtime object as the key and the position of Utility-Info object in Object-Utility as the value.

Freelist is a data structure which maintains the index of free slots in the version-cache. When the system starts it is initialized to maintain the list of all the slots numbers (in the serial order from 1 to cache size). It behaves like a queue, giving the slot numbers as and when required. Freelist is implemented using LinkedList data structure of Java. Each element in the list stores the slot number.

Using the above data structures runtime objects of versions are stored and accessed from version-cache. The next subsection explains the manipulations required in more detail.

3.3.1 Version-Cache manipulations

The sequence of operations performed while inserting a runtime object of a version into version-cache and operations performed while accessing a runtime object from version-cache are discussed in this section.

Inserting a runtime object into version-cache: While inserting a runtime object into version-cache, the exact slot number where the object has to be inserted is obtained from freelist. If the freelist is empty (when version-cache is full), Utility policy is used to get the victim object. The new object is given a utility factor of 1 and compared with the victim object. The victim object (say V_O) is replaced in the cache with the runtime object (say R_O) under two conditions.

1. if the utility factor of V_O is less than utility factor of R_O
2. if the utility factor of V_O is same as utility factor of R_O and frequency factor of V_O is greater than or equal to R_O

The first condition assures that the new runtime object to be placed into the cache is reused more number of times than the victim object.

The second condition is where both the victim object and the new runtime object to be placed have the same utility factor. In such conditions the frequency factor is compared, if frequency factor of victim object is greater than the frequency factor of the new runtime object it is replaced. When the frequency factor of both the objects is same, the victim object is replaced giving more preference to the new object.

There may be cases where all the objects in version-cache have a utility factor greater than one. This indicates that the objects in version-cache will be reused more than once. With the current approach of estimation, the utility factor of a new object that needs to stored in the cache is atmost one. In such cases the new object is not placed in the cache as the objects existing in the cache will be reused more number of times than the new object. The runtime objects are created, used for change detection and discarded after use. This is similar to case when an object cannot be placed in the cache (in general DBMS) and so will be used directly from the main memory.

Once the slot number is obtained, the Runtime-object is created and inserted into version-cache. The object-mapping for the runtime object name to the slot number is

created. Utility-info object is created with the runtime object name, utility factor and frequency factor and stored in the object-utility. The position of utility-info object is maintained against the runtime object name in utility-mapping.

Using object from version-cache: If a runtime object exists in the version-cache, using object-mapping, the slot number in which the object exists is obtained. For the given slot number, the runtime object of the version is retrieved from the version-cache and used. Using utility-mapping, the position of utility-info object in object-utility is obtained. The utility factor of utility-info object is updated and the utility-info objects in object-utility are arranged according to their utility.

Utility policy: Utility policy is designed to ensure that the version-cache always maintains objects that are needed more number of times in the system. Utility policy uses the data structures that are maintained for storing the utility information to decide on objects that can be replaced. The utility-info for each runtime object is updated according to the usage of the object. An object with the least utility is chosen for replacement. If there are more than one object in object-utility with the same utility factor, then frequency factor of the version is considered. An utility-info object *U1* is said to have a lesser utility value than another utility-info object *U2* in two cases:

1. if the utility factor of *U1* is less than utility factor of *U2*
2. if the utility factor of *U1* is same as utility factor of *U2* and frequency factor of *U1* is greater than *U2*

As and when the utility-info of an object is updated, the objects in object-utility should be arranged/positioned in such a way that the object with least utility is easily available. One approach to achieve this is to sort the utility-info objects according to the utility factor whenever an object's utility factor is updated. Another approach would be to maintain object-utility as a min-heap. Apply the minimum heap mechanism whenever a utility-info object is updated. As the requirement for object-utility is to obtain an object

with the least utility factor whenever required, sorting the whole set is not required. The heap approach is adopted here.

The object-utility data structure is implemented as an array of utility-info objects and is viewed as a complete binary tree. Each node of the tree corresponds to an array object with the first object of the array being the root of the tree. Min-heap approach is used to arrange the nodes in the tree. When the min-heap approach is applied over an array, the root node stores the object with the least utility in the system for the given criteria. At any point, the first element of the array represents a utility-info object with the least utility.

3.4 Implementation

The data structures and their implementation details have been explained in the previous section. This section gives a briefing of how the data structures are used along with the system. During the execution of the system from run time objects of versions will be either saved in Version-Cache or used from Version-Cache. The Version-Cache should be accessible from more than one module. Fetch module will access the Version-Cache for inserting the first version, Change detection module will access Version-Cache to retrieve as well as save the run time objects of versions. So the Version-Cache will be accessed from more than one module. A class ObjectManager is introduced which has the following attributes

- CACHE_SIZE - Size of the VersionCache
- Version-Cache - data structure as discussed in the section 3.3
- Object-Mapping - data structure as discussed in the section 3.3
- ReplacementPolicy - an interface to provide replacement behavior

ReplacementPolicy is implemented in a way that the underlying policy can be changed depending on the configuration parameters. *ReplacementPolicy* is implemented

as an interface, methods have been designed in such a way that the basic behavior of a replacement policy can be achieved using them. The following are the abstract methods of the interface

- public abstract String insert (Object obj,int slot);
- public abstract void delete (Object obj,int slot);
- public abstract void update (Object obj,int weight);
- public abstract int getSlot (Object o);
- public abstract boolean isBufferFull ();
- public abstract void init (int size);

ReplacementPolicy is an attribute of ObjectManager class and can be used to get the victim object when the cache is full. This interface can be given the behavior of LRU, FIFO, or Utility policy. UtilityPolicy is another class which implements *ReplacementPolicy* by adding the exact functionality for the above mentioned methods. Apart from these methods it has some attributes, they are

- CACHE_SIZE - Size of the Version-Cache
- Object-Utility - data structure as discussed in the section 3.3
- Utility-Mapping - data structure as discussed in the section 3.3
- freeList - a queue to store the slot numbers that are free in version-cache

The ObjectManager will be initialized with the required size (as per the configuration file) when the WebVigiL system is initialized. The WebVigiL system will also assign the behavior of replacement policy based on the configuration parameters. If utility policy has to be applied, an object of utility policy class will be set to the ObjectManager's replacement policy variable. Similarly if LRU behavior is desired, a class for LRU policy which implements the behavior of replacement policy needs to created and this object is assigned to ObjectManagers variable.

ObjectManager is an static member of WebVigiL system. It can be accessed from other modules. Other modules access the ObjectManager and interact with it for storing and retrieving objects from Version-Cache. If an object needs to be placed, the ObjectManager is accessed and the object is inserted, if the Version-Cache is full, ObjectManager uses the replacement policy to decide on the victim object.

This design allows the replacement policy to be changed when desired by changing just a configuration parameter and adding the required classes. Internally no code has to be changed in ObjectManager or the WebVigiL system. To handle the issues synchronization of multiple threads, locks have been used. The Mutex locks [14] and Read-Write locks were developed by the previous team. The properties required for version-cache (utility factor) are updated every time a runtime object is accessed or saved. Based on these properties the objects are removed, inserted and rearranged. As all the thread accesses result in structural modifications of data structures, mutex locks are used to access to Version-Cache.

3.5 Experimentation results

In this section, case studies have been presented showing the behavior of the version-cache approach and the effectiveness of the utility policy is compared with other policies. Here we have chosen Least Recently Used (LRU), First In First Out (FIFO), and Most Recently Used (MRU). If there are situations where these policies might work well for version-cache replacements, such cases are discussed and compared with the utility policy.

In WebVigiL, requirement of the versions depend totally on the sentinel specifications. There can be sentinels requesting different URLs. In such cases, a version of a sentinel can be reused only once (when a new version is fetched for that sentinel). If there are multiple sentinels on the same URL, same versions may be required by more one sentinel. The behavior of the replacement policies for such situations is analyzed.

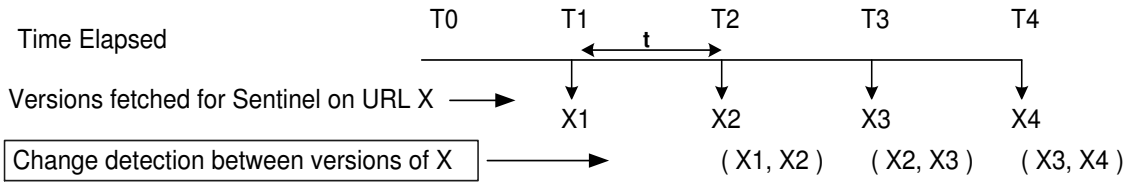


Figure 3.8 Change detection over versions of a sentinel

The first subsection presents the parameters used for comparing the replacement policies. The next subsection gives the scenarios that may occur for WebVigiL and discusses how each replacement policy behaves (with the help of the parameters) in such scenarios.

3.5.1 Parameters for analysis

To analyze the behavior of each replacement policy, some parameters have been introduced. These parameters give an idea of the number of versions parsed and processed in the system and how far the Version-Cache is able to provide the objects for change detections. The parameters are:

1. Total number of pages parsed and processed *(C_T)*
2. Minimum number of pages to be parsed and processed *(C_M)*
3. Number of pages parsed and processed using version-cache *(C_O)*

The example below explains what each parameter represents.

Consider a sentinel which fetches 4 versions of a page at a fetch interval of 't'. This is shown in the figure 3.8, versions are represented as X1 , X2 , X3 and X4.

Change detection will be performed between *(X1 , X2) (X2, X3) (X3, X4)*

From the figure 3.8, it can be concluded that for sentinel S1, the runtime objects for first version X1 and the last version X4 are created only once. For all other intermediate versions (X2 to X3) they are created twice.

C_T would 6 = [1 for first Version + 2 for each Intermediate version + 1 for last version]. This can be generalized for a sentinel.

For a sentinel with K versions, $(C_T) = 2*K - 2$.

Redundant parsing and processing can be avoided by reusing the objects created for intermediate versions. When redundancy is avoided each version will be parsed and processed exactly once, so 4 pages will be parsed and processed (once for each page).

Minimum number of pages, $(C_M) = K$

When an version-cache is used, the effectiveness of the replacement policy used to maintain runtime objects can be determined by the number of pages parsed and processed to perform the change detections for a given situation. Version-cache is used without any replacements when $(C_O) = (C_M)$

The above discussed parameters have been applied for different scenarios to analyze the effectiveness of utility policy.

3.5.2 Behavior of utility policy

To analyze the behavior of utility policy the following scenarios have been considered. Each scenario has been described and experiments have been conducted to observe the behavior of different replacement policies. It is assumed that the *first versions* for all sentinels are saved into cache as soon as they are fetched (only if the cache is not full). For all other versions, they are saved during the time of change detection. If the object exists in the cache they will be used, if it does not and the cache is full then a replacement policy is needed to identify a victim object and replace that object with the new object. The various scenarios are described in this section.

When no sentinels share the same URL:

In a scenario where there are no sentinels sharing the same URL, the versions of a particular URL will not be used by any other sentinel. The maximum reusability or the

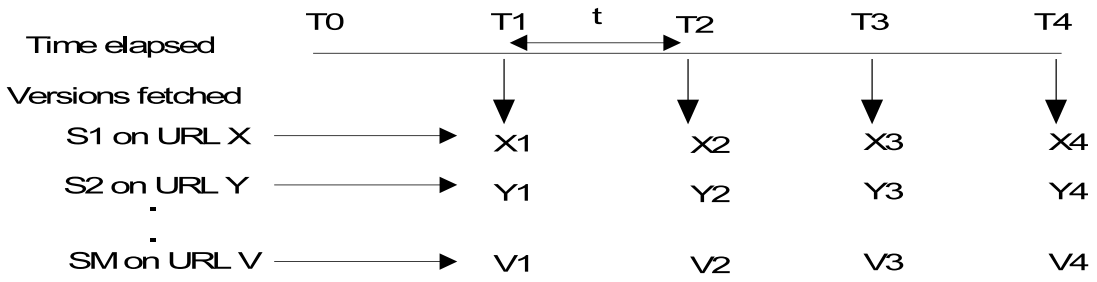


Figure 3.9 Sentinels with unique URLs and same fetch intervals

maximum utility factor of a version is 1. The runtime object of a version is computed when it is required for the first time and will be reused exactly once (when a new version is fetched for that URL). Here there can be two cases for this scenario:

Case 1: When all the sentinels have the same fetch interval. Figure 3.9 shows this case when there are M different URLs (requested by M users) at the same fetch interval. Here S1, S2 represent the sentinels requesting for web pages X,Y,Z,...,V respectively, 't' represents their fetch interval. T0, T1, T2, T3 and T4 represent the time points (fetch points) after every time period of 't' when the versions will be fetched.

As the Figure 3.9 shows, that when sentinels start at the same time, the versions of different URLs will be fetched at the fetch points, according to the interval. Change detection will be performed as and when the versions are fetched. Redundancy will be totally avoided when runtime object of each version is stored in cache and reused. Cache size required for different replacement policies are

- Utility policy requires a cache size of M
- MRU will require a cache size of M
- FIFO will require a cache size of M
- LRU will require a cache size of 2M

In this scenario as the fetch interval 't' is same and the start time is the same, the first versions will be placed in the cache in sequential order in which they were triggered.

For discussion purposes, let us assume that the sentinels are triggered in the order they are represented in the Figure 3.9. The figure also shows the arrangement of the cache with a size of M after the first versions are saved in it.

In utility policy, the object that will be reused is given a utility factor of 1 before saving it into cache. When it is used from cache, its utility factor is reduced by 1. All objects that have a zero utility will be removed from the cache. In this scenario, each runtime object is reused at the most once. So all the runtime objects in the cache will have a utility factor of 1. When change has to be detected between two versions, say X1 and X2, first X1 is used from the cache and its utility factor is reduced by 1. As utility factor of X1 reduces to 0, its slot gets empty. At this point, the runtime object of X2 is created, as this object can be used for performing change detection with X3, it is saved in cache. This object will be saved in the slot created by X2. Similarly, the object of URL Y will use only one slot to store its objects. So at any point, one slot in cache is sufficient for storing the runtime objects of a URL.

In MRU, the object used most recently will be more prone to be replaced. Consider the same case of computing change between X1 and X2, when X1 is used from the version-Cache, it will be more prone to be replaced, when the object of X2 has to be saved, this object is replaced. In this scenario, as the objects in the cache can be reused at most once, X2 will be the one not required again in the system. Even MRU will require a single slot for runtime object of a URL.

In FIFO, the objects will be replaced in the order they are saved. Here the order in which the objects in the Version-Cache are saved is important. In this scenario, as the sentinels start at the same time with same fetch interval, the order in which they are saved will be same as the order in which they are requested. As shown in Figure 3.9, if X1 and Y1 are saved one after the other, in the next fetch cycle, X2 will be fetched before Y2. While performing change detection between X1 and X2, X1 will be used from

cache. When X2 has to be inserted into the cache, the first object (X1) is selected as the victim object. The version-cache will have Y1 as the first object making it prone to replacements. When Y2 is fetched and change detection between Y1 and Y2 is triggered, Y1 will be used from cache. The object of Y2 is created and to accommodate the object of Y2, Y1 is selected as the victim object (as it is the first object in the cache). It can be observed that for this case, victim objects are always the objects which are not required in the system any longer. The version-cache will be able store and provide runtime objects for this scenario without replacing the objects that are required for other sentinels.

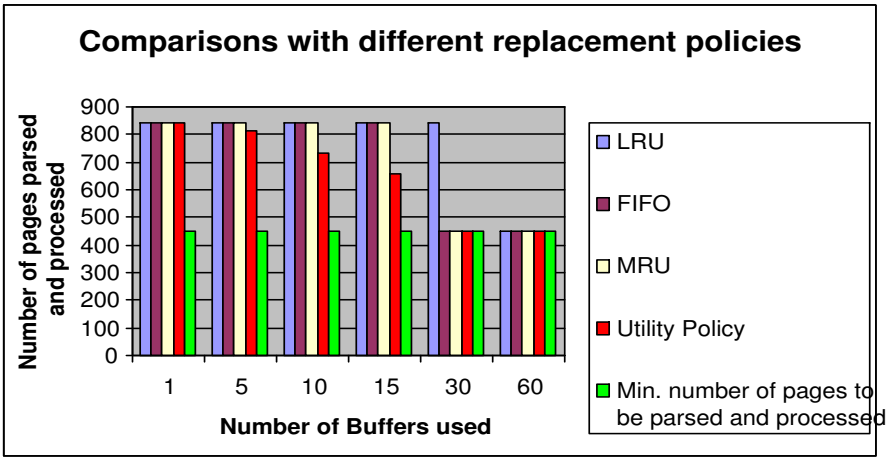
In LRU, the idea is to chose the object which is least recently used. In this approach, when an object is used it is given more priority to remain in the Version-Cache. So when change detection is performed between X1 and X2, and X1 is reused, unlike other policies, it gives the object more priority. To accommodate the runtime object of X2, it will choose Y1 as the victim object. When Y2 is fetched, the object of Y1 is lost and will have to be parsed and processed again. LRU requires a size of 2M to avoid replacing objects required by other sentinels.

Experiments were conducted, by running a set of sentinels according to the given scenario. To analyze the behavior of *Case 1*, tests have been performed on set of 30 sentinels on 30 different URLs with a fetch interval of 1 minute for a period of 15 minutes. Experiments were performed using FIFO, LRU, MRU and Utility policy as the replacement policy for version-cache with different cache sizes. In each run, the number of pages parsed and processed during change detections were logged in files. For a single sentinel, with fetch interval 1 minute for a period of 15 minutes

The number of pages parsed and processed = 28 [2 *15 - 2]

Minimum number of pages to be parsed and processed =15

For all 30 sentinels, (C_T)= 840 and C_M 450



% Improvement of Utility policy over other policies

Cache Size	LRU	FIFO	MRU
1	0	0	0
5	3	3	3
10	13	13	13
15	22	22	22
30	46	0	0
60	0	0	0

Figure 3.10 Comparison of effectiveness of different policies

The figure 3.10 shows the comparison of number of pages being parsed and processed when cache uses different policies (for different cache sizes). In the bar graph, for each cache size the bars represent the number of pages parsed and processed using LRU, FIFO, MRU, utility policy for version-cache, and the last bar shows the minimum number of pages to be parsed and processed if all runtime objects are computed once and reused whenever required for this situation.

It can be observed that C_O=C_M=450 for LRU, MRU, FIFO and utility policy when the cache size is 60 (= 2 * 30). Except LRU, all policies work well for a cache size of 30. The last bar in the graph represents the minimum number of pages to be parsed and processed for the given situation. This is when all the objects are computed once and reused when required again. As the cache size reduces, there will be replacements in the cache so the number of pages that are parsed will increase. This is because there are some versions for which the runtime objects will be created more than once. But even with lower cache sizes utility policy is able to achieve less number of pages to be parsed and processed than other policies. The improvement of utility policy over other policies for different cache sizes is given in a tabular form in the figure 3.10. The improvement

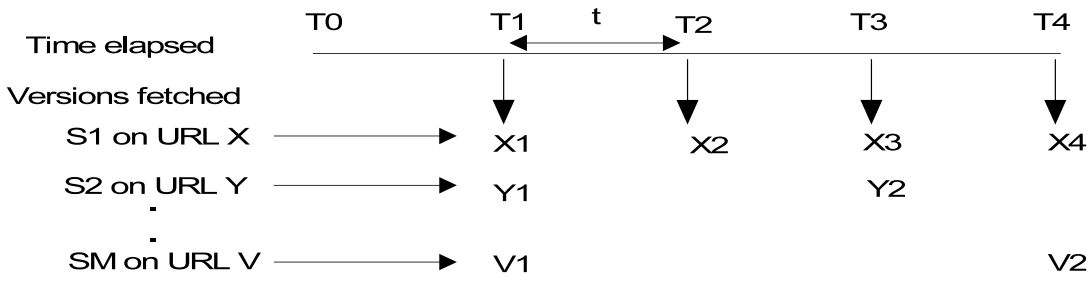


Figure 3.11 Sentinels with unique URLs and different fetch intervals

shown here is in terms of the number of versions parsed and processed (which involves the cost of disk access and page parsing for creating runtime objects). For a cache size of 30, utility policy shows an improvement of 46 % over LRU. This means, using LRU for version-cache if 100 versions have been parsed and processed, using utility policy only 54 versions have been parsed and processed. The cost involved in disk access and page parsings are reduced using utility policy for version-cache.

Case 2: When sentinels have different fetch intervals. Figure 3.11 shows this case when there are M different URLS (requested by M users) at different fetch intervals.

Unlike the previous case, the versions for different sentinels are fetched with different fetch intervals here. Even in this case, a version object is not reused more than once. Cache sizes required for different replacement policies are:

- Utility policy requires a cache size of M
- MRU will require a cache size of 2M
- FIFO will require a cache size of 2M
- LRU will require a cache size of 2M

In this case, the order in which the versions will be required depends upon the fetch interval. As shown in the figure, the versions of URL X are fetched more often than versions of Y. Runtime objects of Y may be more prone to be victim objects when FIFO or LRU is used, as these objects will not be used as frequently as other objects.

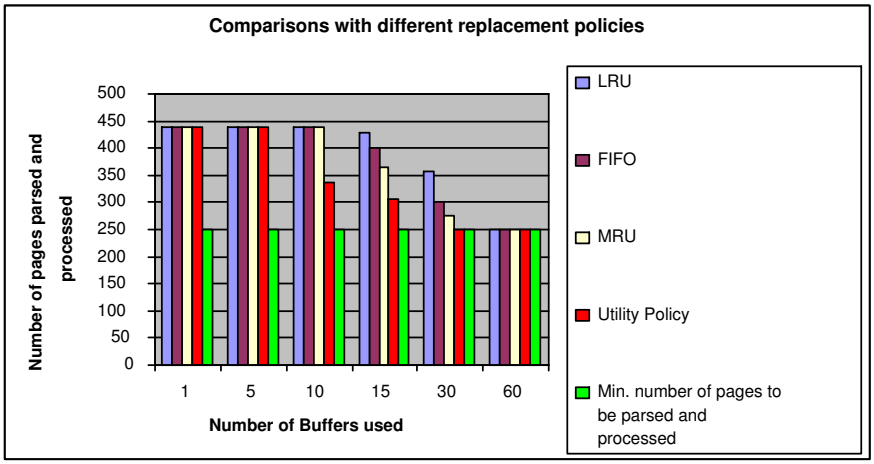
Sentinels with fetch interval	Number of versions fetched for a sentinel in 15 minutes (= K)	Computations for a single sentinel with K versions = (2*K -2)	Computations for 10 sentinels
1 minute	15	28	280
2 minute	7	12	120
4 minute	3	4	40
Total Number of Computations for 30 sentinels			440

Figure 3.12 Number of pages to be parsed and processed

They require a cache size of 2M to use the objects from cache without replacing objects required for other sentinels. MRU performs better than LRU and FIFO when a cache size of M is considered. But it does perform better than utility policy as it replaces the most recently used objects when cache is full. In this case the most recently used object in the version cache may be still reused again. When a cache size on 2M is provided however MRU behaves in the same way as other policies. Utility policy depends on the utility factor. When there are M unique URLs with cache size of M , atleast one runtime object of a version of a URL can be stored in the cache. In this case each runtime object is reused only once. During change detection when an object is used from version-cache its utility factor reduces to 0. When a new object has to be placed it chooses the slot of objects with 0 utility factor. So at any point, one slot in cache is sufficient for storing the runtime objects of a URL.

To validate this behavior, experiments were conducted by running 30 Sentinels over 30 URLs for 15 minutes with different fetch intervals. Among the 30 sentinels, 10 sentinels were with 1 minute interval, 10 sentinels were with 2 minute interval and the remaining 10 sentinels had 4 minute interval.

With sentinels having different fetch intervals, the number of pages that will be parsed and processed for the above setup are shown in the figure 3.12.



% Improvement of Utility policy over other policies

Cache Size	LRU	FIFO	MRU
1	0	0	0
5	0	0	0
10	24	24	24
15	29	24	16
30	30	17	9
60	0	0	0

Figure 3.13 Comparison of the effectiveness of different policies

When the cache is used without replacing objects required by other sentinels, each version will be computed only once and will be reused whenever required. The minimum number of versions that would be parsed and processed for the given set up are C_M for 30 sentinels = 250 [15 * 10 + 7 * 10 + 3 * 10] =250

The figure 3.13 shows the number pages parsed and processed with different cache sizes using LRU, FIFO, MRU and Utility policy. The cache sizes are chosen according to the URLS in the system. Here, as there are 30 unique URLs, M=30. Experiments are conducted for cache sizes 2M, M and M/2. Also the behavior of replacement policies was observed for cache sizes 10, 5 and 1.

In the bar graph shown in figure 3.13, for each cache size the bars represent the number of pages parsed and processed by using LRU, FIFO, MRU, Utility policy, and the last bar shows the minimum number of pages to be parsed and processed for the given setup. It can be observed that for cache size 60 (which 2M), all the policies are able to provide objects with minimum number of disk fetches and processing possible for this case. The figure 3.13 also shows the percentage improvement of utility policy over other policies for different cache sizes.

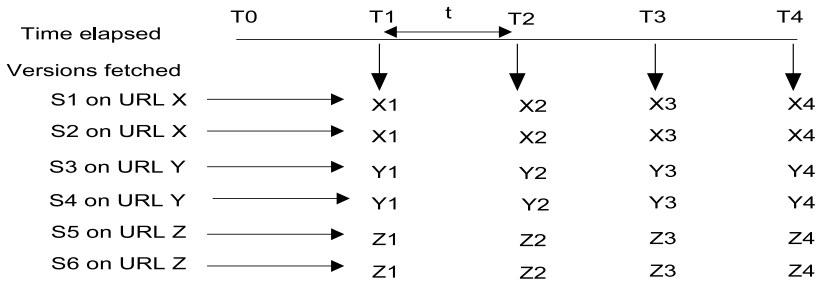


Figure 3.14 Scenario when sentinels share the same URL

LRU, MRU and FIFO decide a victim object based and recency and order. They do not consider any other factors for replacements. For smaller cache sizes this will result in more number of replacements and the objects in the cache will not be reused. Utility policy estimates the utility of a version and also maintains objects in the cache to support more number of change detections. This is the reason utility policy is performing better than other policies even for small cache sizes as shown in the figure 3.13.

When sentinels share the same URL. In the previous scenario shown in figure 3.9, we discussed the behavior of replacement policies where there is no reusability among the versions of different sentinels. Here we consider a scenario where sentinels share the same URL. When sentinels share the same URL, a single version will be reused more than once. The figure 3.14 shows the sentinels S1, S2 accessing the same versions of the URL X and S3, S4 accessing the same versions of Y.

After the time point T1, the cache will have the first versions of all the sentinels, which in this case would be the first versions of each URL. Runtime object of each version is required more than once here. At the time point of T2 when the second versions are fetched, change detection is performed between the second version and first version. The first version can be used from the cache and when the second version has to be saved to cache a victim object to replace would be required. Unlike the previous scenario, the runtime object of first version of the same URL cannot be removed from the cache as

there are other sentinels which need it. The cache size has to be more than the number of URLs in the system to store the objects without any replacements. Cache size required for different replacement policies are

- Utility policy requires a cache size of M+1
- MRU will require a cache size greater than 2M
- FIFO will require a cache size greater M+1
- LRU will require a cache size greater than 2M

In this case, as all the sentinels have the same fetch interval, the versions are fetched in the same order as shown in the figure 3.14. The requirement of the versions also will be in the same order.

Utility policy will assign the utility factors to runtime object of each version. After the first versions are stored in the cache, each runtime object will have a utility factor of 2 (as there are two sentinels which will use the versions of a URL). At time point T2 when the second version is fetched for sentinel S1, change detection is performed between version X1 and X2. The runtime object of X1 will be used from the cache, the utility factor of X1 is now 1, it is still required in the system for S2. To place the runtime object of X2 there needs to be an extra slot in the Version-Cache. If one more slot is provided the runtime object of X2 will be saved here with a utility factor of 1. During the change detection between X1 and X2 for sentinel S2, both objects can be reused from the cache. The utility factor of X1 is reduced to 0 creating a free slot in the Version-Cache. When change is detected between Y1 and Y2, Y1 can be used from the cache. The runtime object of Y2 can now be placed in the slot that is created. Thus, with one extra slot over M slots, the redundancy can be completely avoided for this case.

Given a cache size of M+1, redundancy cannot be avoided using LRU and MRU policies. After time point T2 when change has to be performed between X1 and X2 for sentinels S1, X1 will be used from cache. For X2, the runtime object will be placed in

the cache. For sentinel S2, both X1 and X2 will be used from cache. When change is detected between Y1 and Y2, Y1 can be used from the cache. The runtime object of Y2 now needs a slot in the cache. If LRU is applied here, the slot will be created by removing the object which is least used till then, which would be Z1. If MRU is used, the object of Y1 will be removed. In both the cases, we will losing an object that can be reused. LRU will require a cache size of 2M to for achieving complete reusability. MRU cannot be applied for this situation, even with cache size of 2M it will not be able to achieve complete reusability.

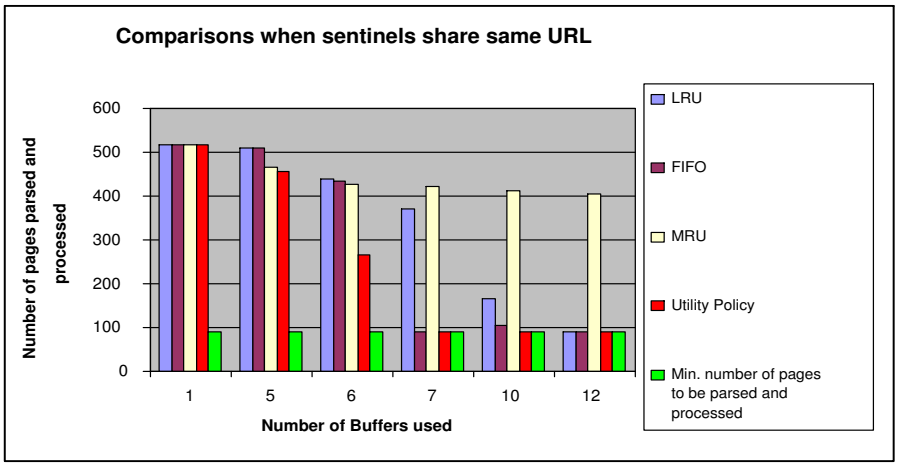
If FIFO is applied in the situation described earlier, X1 will be chosen as the victim object as it is the first object in cache. Since this object is not needed any longer, it can be replaced. For this scenario FIFO will work with cache size of M+1 if the order in which the objects are inserted into the cache remains same the order in which they are required.

To validate this case, experiments were conducted over 30 Sentinels. This time only 6 unique URLS were considered such that 5 sentinels shared the same URL. The fetch interval was taken as 1 minute and were run for 15 minutes. Total number of pages parsed and processed for a single sentinel= 28 [2 *15 - 2]

For all 30 sentinels, C_T= 840

For a URL there will 15 versions fetched in 15 minutes. The same versions will be required for 5 sentinels. The runtime object of a version can be created once for a sentinel and reused for other sentinels. So for performing change detections over 15 versions of a URL (required by 5 sentinels), the minimum number of versions that will be parsed and processed will be 15. As there are 6 different URLs in this case, minimum number of pages that will be parsed and processed for this case are $C_M = 90$ (15 * 6).

The figure 3.15 shows the comparison of effectiveness of different replacement policies.



% Improvement of Utility policy over other policies

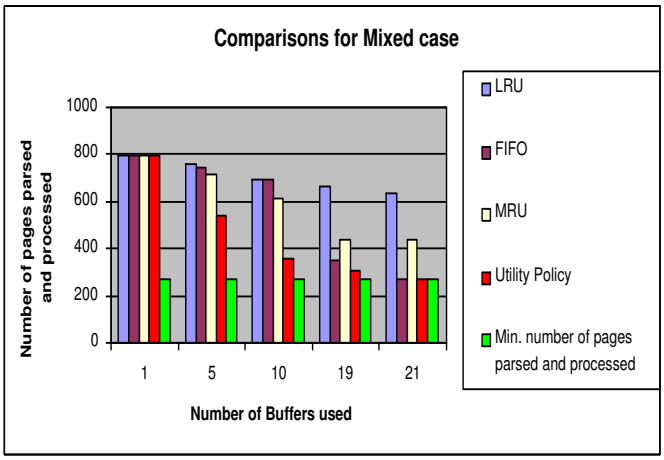
Cache Size	LRU	FIFO	MRU
1	0	0	0
5	10	10	2
6	39	38	37
7	76	0	79
10	46	15	78
12	0	0	78

Figure 3.15 Comparison of effectiveness of different policies

Here as there are 6 unique URLs so M=6, experiments have been performed with cache sizes 12(2M), 7(M+1), 6(M), 5(M-1). From the figure3.15 it can be observed that with utility policy and FIFO the objects in the cache can be used without replacing objects required for other sentinels with a cache size of 7(M+1). LRU needs cache size about 2M and MRU will not be suitable for this arrangement so requires a cache size greater than 2M. In general for different cache sizes, Utility policy is performing better than other policies. The figure 3.15 also shows the percentage improvement of utility policy over other policies for different cache sizes.

A mix of the cases specified above. This is to show the consistency of the utility policy in the scenario where there are some sentinels that share the same URL and also sentinels which have unique URLs. This scenario represents the real time scenario more closely. Based on the experiments performed previously, Cache size required for such a scenario when there are N unique URLs and M URLs are shared by multiple sentinels

- Utility policy requires a cache size in the range of (N+M+1 to N+2M)
- MRU will require a cache size greater than 2N+2M
- FIFO will require a cache size greater 2N+M+1



% Improvement of Utility policy over other policies

Cache Size	LRU	FIFO	MRU
1	0	0	0
5	29	27	24
10	48	48	42
19	54	14	31
21	57	0	38

Figure 3.16 Comparision of effectiveness of different policies

- LRU will require a cache size greater than 2M+2N

Experiments were conducted on 30 sentinels with 15 sentinels or unique 15 URLS and among the other 15 Sentinels such that 5 sentinels share the same URL. So there are 15 unique URLS (= N) and 3 URLs (=M) which are shared by sentinels. The figure 3.16 shows the comparison of different replacement policies for the mixed case and also the percentage improvement of utility policy over other policies for different cache sizes. Here also we can see the same behavior in terms of number of pages parsed and processed. As the cache size is reduced, utility policy requires less number of pages to be parsed and processed. This is because it is able to reuse the pages in the version-cache better than other policies. With the utility factor and frequency factor utility policy is able to maintain the version-cache with the objects that are required the most. It ensures that a new object is inserted only if it has a better or equal utility as the objects existing in version cache. This will reduce the unnecessary replacements to large extent.

3.5.3 Generalization

In the experiments, different scenarios were considered to observe the behavior of replacement policies when there is reusability in the system among sentinels and also when there is reusability only with respect to a single sentinel. It was observed that even when there are situations where a runtime objects can be reused only once, utility policy is able to retain the object in cache till it is used. The other policies depend upon the order and fetch interval as they check for the oldness of the object instead of its reusability. With utility policy we are able to reuse the objects in the version-cache better than other policies in all the cases. Even when the cache size is small, the utility policy carefully retains the objects that are required to reduce more redundant disk fetches and processing. For a given cache size, if the scenarios are in such a way that there cannot be any reusability of the versions then utility policy will behave in the same way as other traditional replacement policies. Thus the policy can be used to maintain the objects in version-cache.

CHAPTER 4

DIFF-BASED VERSION MANAGEMENT

Version Management module of WebVigiL manages a repository of all web pages required for the sentinels in the system. In WebVigiL, the purpose of the version manager is to store the versions for each page and provide them to the change detection module as required. Versions of a page may be required while performing change detection and notifying the changes to users. It assigns a physical directory on the disk [13] for each URL and stores all the versions corresponding to the URL in that directory. Fetch module retrieves a web page in the form of text and provides it to the version manager. Version manager stores this text/ascii file in the corresponding directory. For a given version, the corresponding file in the directory can be obtained from the information that the version manager maintains for each version. Version manager also performs a deletion operation whenever the versions of a URL exceed a threshold. This threshold is a number referred to as deletion trigger and is derived from the sentinel specifications. Deletion algorithm checks for the versions which are no longer required in the system and deletes them. There may be situations where deletion algorithm will not be able to delete any versions. The storage space required to maintain repository of versions of different URLs in such cases will grow considerably. An approach to reduce the growth of the repository as and when the versions are being fetched is desired.

For web pages, the text retrieved by fetch module contains the data along with markup tags which define the structure of the web page. In XML pages, the tags define the nature of the content whereas in HTML they define the presentation aspect. In the current approach, when a version is fetched, the entire text is stored in a file even though

it may be a single line of data that has been added/modified in the new version with respect to the previously fetched version. Storage space required to save the versions can be considerably reduced if the difference between the two versions is stored in the repository instead of the entire text.

Tools like CVS [18] and SCCS [19] use the concept of storing the difference between two files to maintain versions of the same file. These tools maintain entire text of a single file and consider it as the reference file or base version. For all other versions of the file, only the difference with respect to the reference file is stored. When a particular version is required, the contents of that version are generated using the difference file and the base version. The approach here assumes that the versions of a single file do not entirely differ in terms of content and that the size of the difference files is much smaller than the size of corresponding versions.

A similar approach can be applied in WebVigiL. In WebVigiL, the versions of a web page are retrieved over a fixed period of time. Changes in web pages are most likely to be in terms of appearance/disappearance of some content, links or images, as the presentation structure remains stable most of the time. Here it can be safely assumed that storing the difference between the versions of a page would reduce the storage space required for maintaining the repository. However in situations where two versions differ to a large extent, storing the difference may not really reduce the storage space. This chapter presents a diff-patch approach to store only the difference between the pages and handle the situations where the difference files are too large. Unix-based diff and patch utilities are used to compute the difference between two files and regenerate the contents from diff files when required. The section 4.1 discusses the rationale behind using diff and patch utilities.

4.1 Requirements for using diff

Versions of a web page should be stored as difference files, and regenerated when required. Tools are available to accomplish these requirements. Version management module should be able to use these tools during the runtime and store the version as difference files. Further, it should be able to get the original contents of the file from the difference files. CVS and SCCS are Unix-based tools which allow storing the versions of a page.

CVS tool is used to maintain versions of a file which is based on RCS [21]. Internally, CVS always stores the latest version of a file with entire text (as base version) and stores the difference files of previous versions (with respect to the latest version). This is done, as it assumes that the latest version is required more often than previous versions. In this approach, as and when an updated file is added, the existing base version is used to regenerate the original contents for each difference file. The latest version will be considered as the base version and the difference files of the previous versions will be generated and stored. This approach introduces considerable overhead if CVS is used for WebVigiL as the above process should be performed every time a new version is stored. Moreover, control over individual difference files cannot be obtained with the CVS tool. There can be situations when two versions of a web page do differ considerably, and storing the difference file would not save any storage space. To identify such cases and perform required steps, control over the diff files is needed.

SCCS [19] is another tool which provides a similar functionality for storing versions of a file but considers the first version as the base version. It uses the Unix diff utility internally to generate difference files and patch utility to regenerate the actual contents. With the SCCS tool, the handle to individual diff files cannot be obtained. Instead of SCCS, the Unix-based utilities ‘diff’ and ‘patch’ can be directly used. Diff utility can be used over two files to compute the difference and store it in a separate file (difference file).

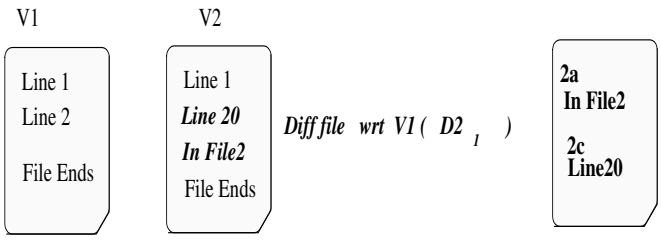


Figure 4.1 Difference between two files V1, V2 using diff

The contents of the original file can be regenerated from difference files when needed. As the differences between files are stored in separate files, handle over the individual difference files can be obtained.

The idea here is to save only the (small) difference between versions of a page to minimize the disk space for storing versions. This is achieved by using the diff and patch utilities in Unix-based systems. In Microsoft Windows environment, the Cygwin environment can be used to handle diff and patch tools. The effectiveness of this approach with respect to storage space reduction and additional time taken have been analyzed. The approach can use any other tools which give the same functionality and control.

4.2 Design and analysis of diff-patch approach

A diff-patch approach is used in WebVigiL to store the different versions of a page. The Unix-based diff utility is used to generate the difference between two files, and the patch utility to regenerate the contents of the original file from the difference file. The diff tool takes two files as input, compares the files line by line and gives the difference between the two files in a readable format. The changes irrelevant to WebVigiL, such as addition of blank lines, blank spaces, case-sensitive differences can be ignored while comparing the two files using the options provided by the diff tool.

Consider two versions V1 and V2 of a web page. Version V2 is an updated version of V1 in which the second line is changed and an extra line is added in V1 to form V2. When the diff utility is applied over the two versions with V1 as the base version, a difference file $D2_1$ is generated. This will be referred as *diff file* in further discussions. The diff file $D2_1$ maintains information to transform the contents of V1 to V2. As shown in Figure 4.1, the file $D2_1$ stores the lines in version V2 that differ from the lines in version V1 and some other meta information. This meta information consists of the line references in the base version, which is required to generate the original contents from the diff file. In the above example, the diff file maintains the new line in V2 and also some information as '2a' to indicate that this line was added into V1 after the second line.

The original contents of a file stored as diff can be generated using the Unix-based patch utility. In the figure 4.1, if contents of V2 can be generated from diff file $D2_1$ and the base version V1. According to the metadata information in diff file $D2_1$, the lines in the $D2_1$ are added, modified or deleted from the base version V1 to generate the original contents of the file V2.

Base Version: Diff files of the versions have to be generated with respect to a version. This is referred as the base version in this context. For the base version alone, entire text of the web page is stored. Every other version fetched will be stored as the diff file with respect to the base version. Every URL will have a base version, the information of base version for a URL should be easily available as it is important while generating contents from diff files. For a web page, the first version fetched is considered as the base version in WebVigiL. When the subsequent versions are fetched, diff is computed with respect to the first version and the diff file is stored in the repository. Whenever the contents of a version stored as a diff file are needed, the first version can be patched with the diff file. The reason for choosing the first file is that it is simple and easy to

maintain. A single piece of information of a base version for each URL can be stored such that it is easily accessible.

One concern for having the first version as the base version is that after a period of time the contents of the versions fetched can be considerably different from the first version. In such cases, the size of the diff file may be almost equal to the original versions fetched. An approach to avoid this is to consider another intermediate version as a base version and store the diff files with respect to that version. But if there is more than one base version, every version stored as diff file should maintain the information of its base version. To avoid these complications the first version fetched for a URL is considered as the base version for all the versions fetched for that URL. However, when the deletion algorithm of version manager is triggered, the base version will be changed. Handling base version and diffs of other versions is discussed in later sections of the chapter.

Size of diff files: The diff tool compares a given version, say V_X, with the base version V_B line by line and stores the lines that are different from the base version along with some meta information. The size of the diff file approximately depends upon the number of lines in V_X that are different from V_B. Size of the diff file is not always less than the size of file V_X. if the file V_X is completely different, i.e., all the lines differ from the base version, then the size of the diff file will be almost equal to the size of V_X or may even be more than V_X. It can be observed that

1. if V_X is a file which has x% of the lines different with respect to base version V_B and DX_B is the diff file computed between V_X and V_B then

   Size of DX_B = *x% (size of the file* V_X *)*

The patch tool uses the information in diff files to regenerate the contents of the original file. As the diff file's size increases, it will have to process more lines to regenerate the original contents. The diff size may affect the computation time involved in regenerating the original files. The next section presents a study of the behavior of diff

sizes with respect to the time required for computing the diff and generating contents from diff files. Web pages can be of different sizes and it is observed that most of the commercial sites maintain a size of 40 kilobytes to 60 kilobytes of data. Relatively large pages can have a size of 80 kilobytes to 100 kilobytes; also some web pages such as course websites where there is relatively less amount of data (and no graphics or popups), the pages size ranges from 20 kilo bytes to 30 kilobytes. To analyze the computation cost of applying diff and patch, pages of each category have been considered.

4.2.1 Analysis of time taken for diff-patch aproach

This section presents the analysis of the computation complexities involved in maintaining and using the diff files. With diff-patch approach when a page other than the base versions (first version) is fetched, diff has to be computed and stored. Further when the version is required, the original contents are regenerated. Some additional computation time is incurred each time contents of a version are stored or retrieved.

Generation of Experimental Data: The concern here is to observe the effect of different sizes of diff. Size of diff files can be almost equal to original files or 50% of the size of original files and so on. For a given page, data has been created such that the diff files of different sizes can be generated.

For example, to analyze the effect of diff file that is half the size of actual page, a page P was considered and another page P_d has been created by changing 50% of the lines of page P. The size of the diff file obtained by computing diff of P_d with respect to P was observed to be approximately 50 % of the size of file P_d. To analyze the effect of such diff files(which are 50% of the size of original) 30 P_d pages were created. Experiments to compute diff and regenerate actual contents using these diff files were performed and the average time taken for computing diff and regenerating contents (patch) over 30 pages was computed. Similarly when the effect of diff file that is 10% of the actual page

Diff Behavior for a file of size 80.94KB		
Size of the base version 80.94KB		
% of lines changed in base version	Size of the files generated in Kilo bytes	Size of the diff files wrt to base version
10	81.31	7.40
20	81.79	19.33
30	82.27	26.74
40	82.74	34.30
50	83.22	41.28
60	83.70	48.12
70	84.28	56.41
80	84.92	65.04
90	85.56	74.53
100	86.19	86.20

Figure 4.2 Files created for different diff sizes

was desired 30 P_d pages were created by changing 10% of the lines of P and similar experiments were performed.

Such simulation and experiments have been performed over pages of size 80KB, 50KB and 18KB.

Analysis for a page of size 80KB:

For page of size 80KB, different data sets for generating diff files of different sizes have been created. Figure 4.2 shows the pages generated and the diff file sizes with respect to the base version (which is of the size 80KB).

In the Figure 4.2 it can be observed that by changing (adding content to) 10% of the lines in the base version a file of size 81.31 KB is obtained. The size of diff file obtained by computing diff for this file with respect to base version is 7.40KB which 10% of the size of the file (of size 81.31KB). 30 such pages were generated and the average time taken to compute the diff files was observed. The average time taken to generate the

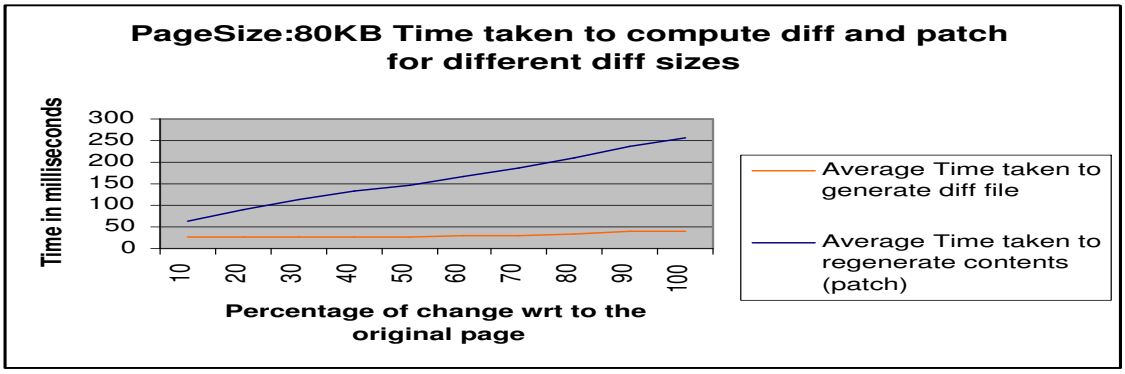


Figure 4.3 Comparison of time taken to compute diff and patch

Diff File Size	% of Improvement in space	% of additional time incurred
10 % of original size	90.9	58.46
20 %	76.37	70.79
30 %	67.49	76.79
40 %	58.55	79.10
50%	50.39	81.08
60 %	42.51	82.04
70 %	33.08	84.62
80 %	**23.41**	**83.25**
90%	**12.89**	**83.54**
100%	**0.01**	**84.11**

Figure 4.4 Improvement in space and additional time incurred for diff

original contents from the diff files and the base version was also observed. In a similar fashion, the time taken for different percentages of diff files is presented in the Figure 4.3

The Figure 4.3 shows the graphical representation of the time taken for computing diff and regenerating original contents for different sizes of diff files. The Figure 4.4 shows the percentage of improvement in storage and the percentage of additional time incurred while dealing different diff sizes.

From the figure 4.4, it can be concluded that for the diff files which have a sizes more than 80 % of the original file, the diff file can be discarded and the original file

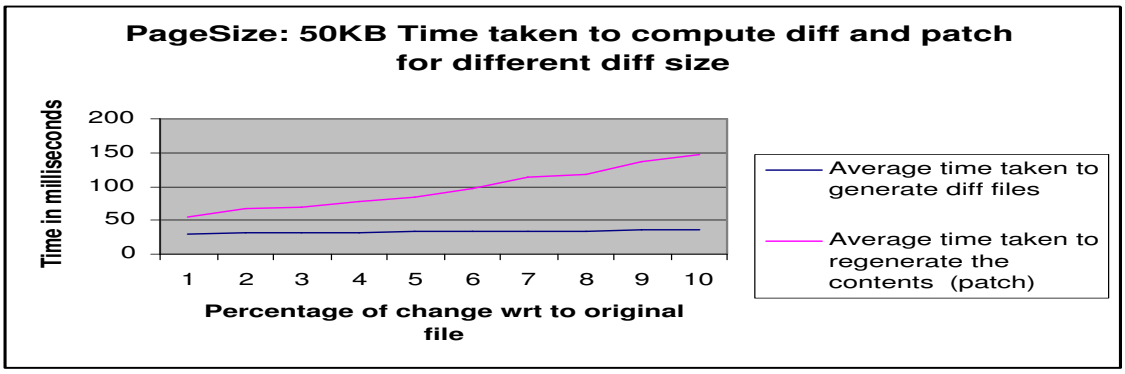


Figure 4.5 Comparison of time taken for computing diff and patch

itself can be stored in the repository. It may not useful to store the diff file as the time incurred in regenerating its original contents is relatively high and the space efficiency obtained is low.

Analysis for a page of size 50KB: A page of size 50KB has been considered as the base version and the data sets required for generating different sizes of the diff files have been created. The figure 4.5 below shows the comparison of the time taken to compute the diff and the time taken to generate the contents from the diff files of different sizes.

The files that have been generated as data set and the corresponding diff files generated with respect to the base version as specified in the figure 4.6. It also shows the percentage of improvement in space and the percentage of additional time incurred in regenerating the contents from the diff files.

From Figure 4.5 and Figure 4.6 it can be observed that even for the pages of size 50KB, the time taken for regeneration of contents is relatively higher for large diff files.

Analysis for a page of size 18KB:

Similar experiments have been performed for a page of size 18KB. Figure 4.7 shows the comparison of the time taken for computing the diff and regenerating actual content from diff files for different size of the diff. The Figure 4.8 shows the sizes of the data set

Diff Behavior for a file of size 50.681Kilo Bytes (KB)				
Size of the base version 50.681KB				
% of lines changed in base version	Size of the files generated in KB	Size of the diff files wrt to base version in KB	% of Improvement in space	% of additional time incurred
10	50.87	4.27	91.60	46.30
20	51.14	12.04	76.46	53.73
30	51.42	13.97	72.82	55.71
40	51.69	17.93	65.30	58.44
50	51.95	22.66	56.37	60.71
60	52.24	29.60	43.34	65.63
70	52.51	35.38	32.61	71.05
80	52.78	41.68	**21.03**	**70.94**
90	53.06	47.33	**10.79**	**73.72**
100	53.30	53.31	**0.02**	**75.68**

Figure 4.6 Behavior of different diff files for a 50KB page

and the corresponding diff files generated with respect to base version. It also shows the percentage of improvement in space and the percentage of additional time incurred in regenerating the contents from the diff files.

The figures 4.7 and 4.8 show the same behavior, the percentage of improvement in the space is relatively less and the percentage of additional time incurred in regenerating contents from diff files is relatively more as the size of the diff files increase. But as the file is relatively smaller than the previous pages, even when the diff file is 50% of the original size the space efficiency gained is less.

Conclusions on maintaining the diff files: From the above analysis, it is evident that as the size of the diff increases, the time taken to compute the diff is almost same, the time taken to generate the actual contents increases and the percentage improvement in space decreases. Further in all the above cases when diff files size is equal to or more than 80% of the original page size, the relative space gained is far less and the relative

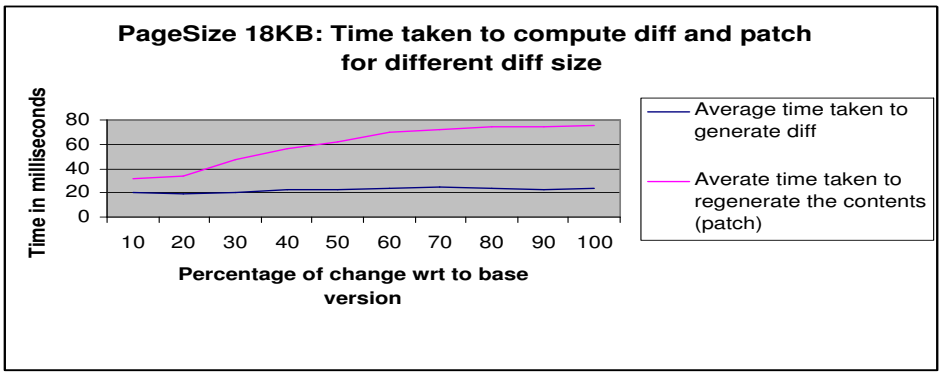


Figure 4.7 Comparison of time taken for computing diff and patch

Diff Behavior for a file of size 18 Kilo Bytes (KB)				
Size of the base version 18KB				
% of lines changed in base version	Size of the files generated in KB	Size of the diff files wrt to base version in KB	% of Improvement space	% of additional time incurred
10	18.47	1.48	91.96	37.50
20	18.51	3.99	78.44	44.12
30	18.56	5.25	71.70	57.45
40	18.60	10.68	42.57	58.93
50	18.66	13.08	29.89	64.52
60	18.73	14.58	22.15	65.71
70	18.80	16.10	14.35	65.28
80	18.86	16.98	**10.00**	**67.57**
90	18.93	17.86	**5.65**	**68.92**
100	19.00	19.01	**0.05**	**68.42**

Figure 4.8 Behavior of different diff files for a 18KB page

time incurred in regenerating actual content is more. As the time taken to compute the diff is more or less the same for different sizes of files, the diff file can be computed and the size can be compared with the original page size. The diff file can be deleted and the original file itself can be maintained in the repository if the diff file is greater than or equal to 80% of the original page size. This value of 80% is considered as a threshold value and is a configurable parameter. Currently it is set to 80% and used for maintaining the versions using diff-patch approach.

4.2.2 Diff files during deletion

Version manager performs a deletion operation over the versions of page to purge the versions that are not required. It maintains some information, based on the specifications of sentinels requesting the URL, and triggers the deletion algorithm using this information. When deletion is triggered for a URL, the version manager performs some operations to identify the set of versions that will be required for different sentinels in the system. It basically identifies the oldest version required in the repository. If the oldest version required is the first version then no version is deleted. But if any other version is identified as the oldest version, it deletes the versions older than this version. In such situations the base version may be one of the versions that has to be deleted. Among the versions in the repository there might be diff files generated for versions with respect to that base version; the contents of these versions have to be generated before deleting the base version. Also the next suitable version has to be made the base version for the URL. The approach followed to handle this situation is explained with the help of an example.

Consider a scenario where the repository of a URL page has 20 versions. Assume that according to specifications the deletion trigger for the URL is 20, which means that the deletion algorithm is triggered when the next version is fetched. Figure 4.9 shows a snapshot of the directory storing the pages. The directory has 20 versions, few stored as the diff files and others in the normal form (stored as entire files). The base version of the URL is V1 in this case.

When a next version, say V21 (V_N) is fetched, the deletion process is triggered for the versions of that URL (www.uta.edu). If the deletion algorithm identifies the tenth version (V10) as the oldest version(V_O) that is required, the first ten versions including the base version have to be deleted. There can be some versions (which are not being deleted) in the directory stored as diff files with respect to the base version. Before

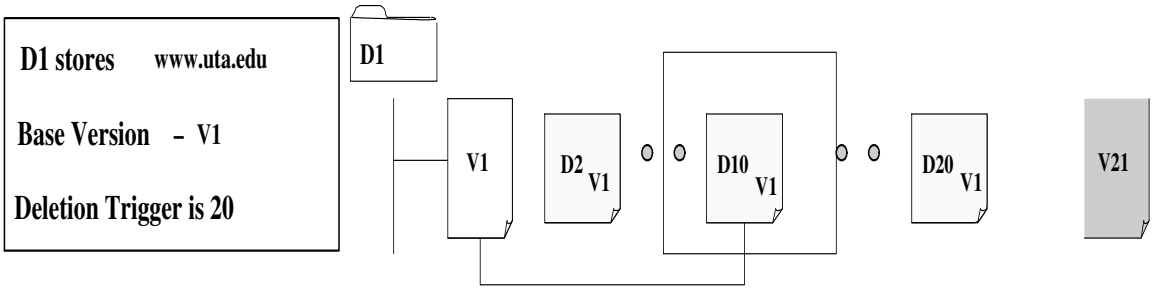


Figure 4.9 Directory of versions with diff-patch approach before deletion

deleting the base version, it is essential that the original contents for these diff files have to be generated. Further, a new base version has to be identified from the next set of versions.

For the above case, the files for the versions V2 to V9 are deleted from the disk. The current base version (V1) is used to regenerate the original contents of versions from V10 to V20 (if they are stored as the diff files in the repository). Current base version V1 is deleted and newly fetched version V21 is made the base version. For all the versions from V10 to V20 (which is like V_O to $V_N - 1$, diff files are computed with respect to new base version V21.

Generalizing the above case, let the base version of a URL is V_B and on arrival of a version V_N, let the deletion process be triggered. Version manager identifies the oldest version V_O, If V_O is the first version in the directory no action is performed. But if V_O is any other version that is not older than V_B (which means base version also has to be deleted), then except for V_B all the versions till V_O (full versions or diff files) are deleted. Using the base version V_B the actual contents of the diff files of the versions from V_O to the version V_N (if any) are regenerated and the diff files (wrt to V_B) are deleted. The current base version V_B is deleted and the version V_N is considered as the base version for the next set of versions. V_N is considered as base version as the new versions that

will be fetched will be similar to the version V_N (the sizes of diff files of new versions will be less) than any other older version in the repository.

4.3 Experiment evaluation for a repository

Experiments have been performed to observe the storage space that can be saved if the diff-patch approach is used for maintaining the repository. The repository size for storing versions of different web pages for three days with and without diff-patch approach has been compared.

4.3.1 Experimental Setup

Data from the web has been collected by fetching different web pages or urls). Sentinels were placed on different URLs with one hour fetch interval for three days and stored in a repository. This repository has been used as the source for running experiments to observe the time latency involved and the storage space gained by using diff-patch approach. As the data has been fetched for three days the maximum number of versions fetched for a given URL can be 72. However there might be not 72 versions fetched for all URLs. In the simulated environment, sentinels have been created on the same set of URLS with one minute as fetch interval for a span of 72 minutes. Instead of fetching versions of URL pages from web, the fetch module retrieved versions of a given URL from the repository and stored them in temporary repository. For cases where there are less than 72 versions, the fetching rules are terminated and the sentinel is stopped if the required version does not exist in the repository. The reason for performing a simulated experiment is to compare the sizes of directories (while saving the versions in repository) with and without diff-patch approach.

During simulation the size of the temporary repository, computation time for storing and retrieving pages have been logged at regular intervals. This log is used for

URL Page	Directory Names	Size without Diff in KB	Size with Diff in KB	%improvement in storage space
http://www.uta.edu	D1	1304	65	95
http://www.hotwire.com	D2	5884	1284	78
http://www.ubid.com	D3	2356	506	79
http://www.rediff.com	D4	1258	133	89
http://www.usnews.com	D5	2768	437	84
http://www.edealinfo.com	D6	5458	2108	61
http://www.cnn.com	D7	4423	2380	46
http://www.anandtech.com	D8	3110	874	72
http://www.stocks.com/	D9	3711	423	89
http://news.bbc.co.uk/default.stm	D10F2	4488	613	86
http://www.football.com/index.shtml	D11F3	4647	269	94
http://www.mtv.com	D12	5223	834	84
http://www.dallasnews.com	D13	8656	918	89
http://news.yahoo.com/	D14	3734	1421	62
Repository Size in KB		57021	12265	**78**

Figure 4.10 Figure showing the size of each directory

analyzing the behavior of the directory sizes and the time taken for storage and retrieval. With simulation, the log can be generated for same set of versions for the URLs with and without diff-patch approach. The Figure 4.10 shows the URLs in the repository, the corresponding directories that store the versions. The figure also gives the storage space required for each directory with and without diff-patch approach.

From the above figure it can be observed that a considerable percentage of improvement in the storage space using the diff-patch approach. Figure 4.11 shows the comparison of the sizes for each directory.

The least amount of space is saved for the URL page www.cnn.com (directory D7) which is a 46% of improvement in storage space by applying diff-patch approach. For all the web pages considered here there is an overall improvement of 78% in the storage space. The size of repository has been logged at regular intervals to observe the behavior of its size. Figure 4.12 shows the behavior of the repository with and without diff-patch approach after every 6 minutes. As the simulation is performed for 72 minutes, with a 6 minute interval the size of repository at the start of simulation and end of simulation can be obtained. It can be observed that as the versions in the repository increased, the

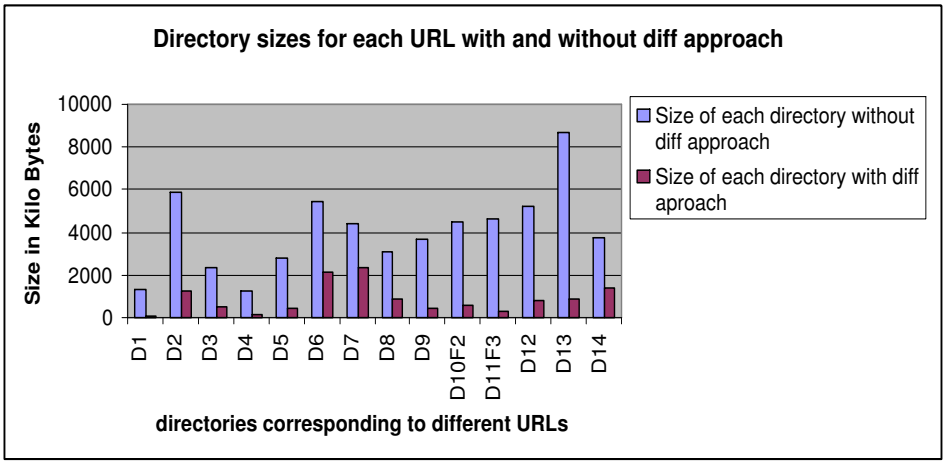


Figure 4.11 Size of each directory with and without diff-patch approach

storage space growth of repository using diff-patch approach is relatively much less than the storage space growth of repository without diff-patch approach.

The time complexities involved in storing the versions and retrieving the versions with and without diff-patch approach have been analyzed. The time taken to store and retrieve a version will involve some additional time when diff-patch approach is applied. This additional time has been analyzed by computing the time required for *storing the page* and time required for *performing change detection.*

Storage Time Storage time is the time taken to store a page in the repository. This involves I/O operations to store the contents of a page in file on the disk. The time for storing a page will be different if the diff-patch approach is applied in the system as there is some additional time for creating diff. For the two situations the storage time can be defined as follows

Storage time when diff-patch approach is not used= Time taken to store the contents in a new file

When the diff-patch approach is applied then a given page has to be stored, diff has to be computed and accordingly either diff file or the page is deleted

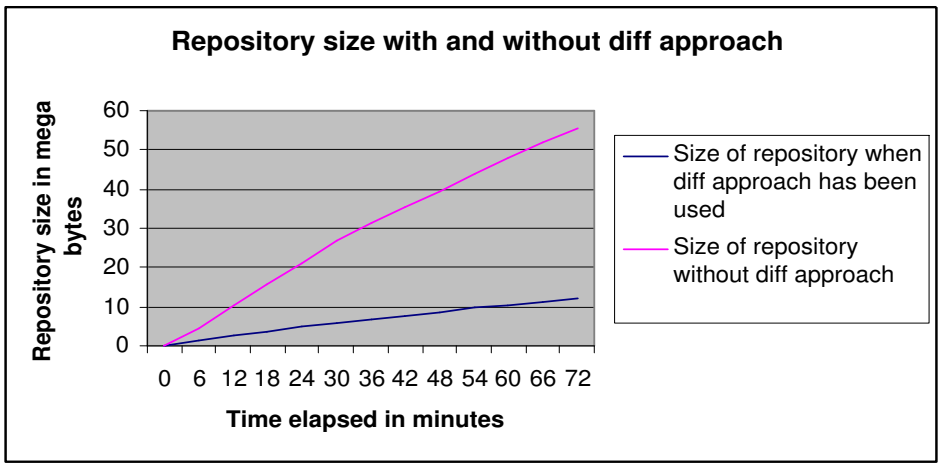


Figure 4.12 Repository of versions with diff-patch approach

Storage time when diff-patch approach is used = Time taken to store the contents in a new file + Computation time for generating diff file + Deleting the new file or diff file based on the size

Computation time for performing change detection While computing change detection between two versions of a URL, the contents of each version are required. In the normal execution without diff-patch approach, the contents from the versions can be obtained from the corresponding files. If diff-patch approach is used to store versions and the versions required are stored as diff files then the original contents have to be regenerated and stored in temporary files. Once the change has been computed these pages have to be deleted. Computation time for computing change between two versions V1 and V2 would be

Computation time when diff-patch approach is not used= Time to retrieve contents from corresponding files + Time taken for change detection

When the diff-patch approach is applied the time would be the same as above if the files corresponding to the versions are available in the repository as full versions. But if

Comparison of computation times (All measurements of time are in milliseconds)								
Directory Name	Avg Time for **Storing** with diff	Avg Time for **Storing** without diff-patch	**Additional Time** incurred while Storing with diff-patch	**% increase in time for storing** due to diff-patch approach	Avg Time for **Change Detection** with diff-patch	AvgTime for **Change Detection** without diff-patch	**Additional Time** incurred for Change Detection with diff-patch	**% increase in time for change detection** due to diff-patch
D1	67	15	52	347	171	68	103	151
D2	76	21	55	262	530	207	323	156
D3	64	16	48	300	258	117	141	121
D4	73	16	57	356	305	135	170	126
D5	81	18	63	350	402	183	219	120
D6	93	39	54	138	1236	714	522	73
D7	83	26	57	219	566	213	353	166
D8	74	17	57	335	351	140	211	151
D9	71	20	51	255	327	132	195	148
D10F2	77	19	58	305	390	165	225	136
D11F3	78	21	57	271	381	152	229	151
D12	68	24	44	183	418	187	231	124
D13	90	40	50	125	864	488	376	77
D14	70	19	51	268	438	192	246	128

Figure 4.13 Comparison of the time taken with diff-patch approach

the files corresponding to versions are stored as diff files then the original contents have to generated and stored in temporary files. In such cases the computation would be

Computation time when diff-patch approach is used= Time to generate contents from diff files corresponding to each version and store in temporary files + Time taken for change detection + Time to delete the temporary files from repository

According to the expressions above the computation complexities have be computed. The figure 4.13 shows the observations of the time taken for storage and computing a change during the simulation (the time here is in milliseconds).

It can be observed that the time taken with diff-patch approach is relatively more than the time taken without diff-patch approach. This is true for both the storage time and change detection time. Also the percentage increase interms of time for storing the pages and performing change detection are showing in the figure 4.13. The maximum percentage of increase in time taken for storing pages is 356 and the average percentage of increase is around 250. Similarly the maximum percentage of increase in time for

computing change detection is 166 and the average percentage is 130. It can be observed that the time taken for computing change detection is more than time taken for storing but the percentage increases are less. This is because the time taken for the change detection phase over two full versions of a URL remains the same in two approaches. The additional time is taken with diff-patch approach to regenerate the contents and delete them after usage. In this case though the percentage of increase is high it can be observed that the actual increase is only interms of milliseconds. From the experiments it can be concluded that storage space growth of the repository can be reduced by applying diff-patch approach consuming less amount of additional time (increase is only interms of milliseconds). In a general case, depending upon the actual increase in time and the percentage increase in time the diff module may or may not be used with WebVigiL. The next section discusses the implementation of the diff-patch approach.

4.4 Implementation

Diff-patch approach has been implemented using diff and patch tools provided by Unix. Java provides functionality for executing external utilities and applications from within a Java program. The *Runtime class* in Java API provides a method *Runtime.exec()* to execute the commands from a java program.

To interpret shell meta-characters such as redirection characters and pipes a shell has to be invoked and the required command has to be passed for execution. The exec method of Runtime class can take a single string as a parameter or an array of string. If an array is given, each element in the array will be executed as a separate command. Here, an array of commands is passed to the execute method. For example to generate diff between two files this is how the commands are executed using Runtime class

String[] command=”sh”, ”-c”, ”diff -abweBi file1.txt file2.txt ¿ diffFile.txt” ;

Runtime.getRuntime().exec(command);

In the above code, an array of commands is created and executed using Runtime. The array's first element is always a command to invoke a shell (in Unix-based systems). Then the second element '-c' is to inform the shell that the next elements have to considered as commands. In the third element, the required command is specified. Here for computing diff it would be to have the base version as the first parameter and then the version for which the diff file is needed. The options in the form of '-abeBwi' is to ensure that while computing diff file insignificant changes like the blank spaces in the line, empty lines and the case sensitivity are not considered as change. The diff file obtained from the diff utility is redirected a file diffFile.txt.

In the similar fashion the for regenerating the contents of a file from the diff file and base version the following code is used

String[] command="sh", "-c", "patch -e file1.txt diffFile2.txt -o newFile.txt " ;

Runtime.getRuntime().exec(command);

In this code, the patch utility has been used to generate the original contents. Patch utility needs the original file (base versions) and the diff file to generate the contents. The base version is updated by default unless until an '-o' is used. This is specified to inform the patch utility to generate the contents and store in a file called newFile.txt

If the diff-patch approach has to work on Windows systems, then required utilities can be obtained by installing Cygwin tool. This tool provides the Unix-based utilities for windows system. In such cases the first element in the command array should be different. Instead of having 'sh', it should be 'cmd.exe' or and equivalent command to invoke the command prompt of windows.

A new class *VersionOrganizer.java* has been introduced into *version manager module*. The class *VersionOrganizer.java* has the methods to compute the diff files and generate contents using java Runtime API. Version manager module uses this class whenever needed. All other modules interact with version manager module. Version manager has

a data structure *latest hash* to maintain the latest version of each URL. This data structure has been modified to store the value of base version of each URL. Latest hash was implemented as a hashtable which stores the URL mapping as key and the latest version object of a URL as value. The value part of the hashtable is now extended to store the information of base version and the latest version object.

4.5 Summary and Conclusions

The diff-patch approach discussed is applied for maintaining the repository of versions. When the first version is fetched for a URL, it is stored as the entire page and is considered as a base version. For all other versions, the contents are first stored in a file, the diff is computed with respect to base version. The diff file is retained and the new file created is deleted if the size of diff file is less than x% of the size of new file. Otherwise, the diff file is deleted and the new file is stored. The value of x is a configurable parameter, for the current system it is fixed as 80. This value for x has been fixed after conducting experiments on different file sizes.

During the runtime of the system, if contents of a version are required, the corresponding file is checked in the repository. If it is not available then using the base version and corresponding diff file the original contents are generated and a temporary version file is created. This version file is deleted after it is used. When the version manager triggers deletion process the required operations are performed and the latest version is made as the base version for the next set of versions. Experiments have been conducted on a set of web pages to observe the computation and disk space consumed when diff-patch approach is applied in WebVigiL to maintain the repository. It was observed that a significant percentage of improvement in space can be achieved incurring increased computation time. In this experiment, the version-cache described earlier has not been used. We believe that the computation time will reduce further when the version-cache

is used in conjunction with diff-patch. Thus, WebVigiL can be scaled up to handle large number of URL's using this approach.

CHAPTER 5

RECOVERY OF WEBVIGIL

The WebVigiL system is vulnerable to system crashes. It is therefore important to address the mechanism of recovery and the issues involved in the same to have in place a fault-tolerant and reliable system for change detection and notification. The limitations of the current system have to be analyzed and means to circumvent the same to handle failures is required. In this chapter, we discuss the short-comings of the current system that limit its capability to handle failures. Also outlined are the requirements for effecting a fair recovery and the design of the recovery module for WebVigiL.

5.1 Limitations of the current WebVigiL

The current WebVigil system supports specification, management, and propagation of changes as requested by a user. The existing system does not start as a separate application, but does so only on the placement of a sentinel by a user. The modules are then initialized on the same address as the servlet engine. This approach makes it difficult to detect failures in the system. Furthermore, the system is at a risk due to the failures that can occur to the servlet engine.

A system failure in the current set-up results in the termination of active sentinels and all change monitoring to requested pages is halted. Also, the runtime information maintained is lost in case of a system crash. The requirement therefore, is to enable the system to swiftly detect failures and perform the requisite steps to restore it to a consistent state. In the following section we discuss the requirements for building a fail-safe and recoverable WebVigil system.

5.2 Requirements for recovery in WebVigiL

To perform recovery, the failure of the system has to be identified, and the system needs to be restored to a stable state. All sentinels that were active prior to the failure need to be identified and re-started. Additionally, runtime information lost during the system crash, also needs to be retrieved. The WebVigil system stores the runtime data structures in memory for handling the storage and retrieval of versions in the repository and for maintaining change detection graphs. This information should be persisted so that the state of these data structures before the crash is available at the time of recovery to restore the system to a stable state.

The version manager maintains [13] the information about versions of each URL page stored in the repository. Every URL is mapped to a unique directory on the physical storage. Data structures to store the mapping information (directory name) for each URL are maintained. This mapping information is required to ensure correct storage and retrieval of the versions. The information corresponding to a version of a URL page fetched is maintained in the version object. Each object stores the version number, the last modified time of the URL page, the value of the checksum and the mapping name of the URL. The versions objects are maintained along with the previously fetched versions in hashtables depending on the fetch rules. During system recovery, the information stored in the data structures is crucial for computing change between appropriate versions.

Also, the change detection module maintains data structures that contain information regarding sentinel grouping. It also stores the relationship between the sentinels placed and URL page they have been placed on, using a graph [14] representation. This graph is required for efficient change detection. The sentinel-related information is persisted in a knowledge base on oracle. During recovery, the sentinels that require to be

restarted have to be identified using the information present in the knowledge base and the required events and rules have to be created.

The WebVigiL system has to be started as an independent application so that failures to the system are detected and the recovery process can be triggered in the event of failure.

5.3 Persisting runtime data structures

The runtime information that is sufficient and necessary to reconstruct the system prior to a crash has to be persisted, or in other words, stored. It is important that the contents of the data structures are stored as and when they are updated. Storing this information in a database will cause an overhead as the database server needs to be accessed for updates. Java provides mechanisms of Serialization and De-Serialization [22] to persist data structures and reconstruct them when required. Serialization is provided by Java to easily write entire objects and primitive data types. It is a mechanism to flatten objects into a stream of bytes that can be written to disk. De-Serialization is a mechanism to reconstruct the object from the serialized byte stream. Since the time of crash cannot be predicted, every update to the data structures has to be persisted. Using such an API would require the system to serialize the entire object each time an update is made. This would be an overhead when the data structures increase to a considerable size. Another approach to persist information is by logging it to a flatfile. Write Ahead Logging (WAL) has been adopted to store only the updates to data structures. In the WAL approach, prior to updating a data structure, the information necessary and sufficient for recovery is written to a log.

The data structures required for grouping sentinels and the change detection graph depend upon the sentinels that are active in the system. Following the crash, these data structures should be built only for those sentinels that were active prior to the time of the

crash. Version-Caching and diff-patch approach also maintain some information using runtime data structures. Version-cache has data structures to store the runtime objects of the versions and also information required for maintaining these runtime objects. On recovery, if the sentinels that need to restart are identified and restarted using ECA rules [9, 10], the required data structures for grouping and maintaining the change detection graph will be created. As and when the versions are required by change detection module the runtime objects and the corresponding information for version-caching can be rebuilt. Thus, there is no need to persist the information regarding these data structures.

The mapping information and the information of versions in the hashtables cannot be rebuilt based on the sentinels. The base version information of each URL is based on the version information and so cannot be rebuilt unless persisted. Hence, the information in data structures maintained by version manager needs to be persisted.

Write Ahead Logs are used in WebVigiL to log the runtime data structures. The data structures grow and shrink during runtime according to the sentinels and their properties in the system. The data structures can be updated at different instances during runtime. As the time of failure is not known, data structures have to be persisted for each update to their contents. Rather than persisting the information in the entire data structure, only the information that reflects the change made to the state of a data structure is logged. The mapping information and version information are persisted by logging the updates to data structures which maintain this information. The information corresponding to the versions is dependant upon the mapping information, hence, the data structures for mapping have to be reconstructed before reconstructing the version information. Separate logs are maintained for the information regarding the mappings and for that regarding the versions.

Mapping log is used for logging the updates to data structures required for mapping information. *Version log* is used for logging the information of a version being inserted

into either best effort hash or fixed interval hash. The information that is required to identify the versions that have been deleted from hashtables is maintained in another log called *deleted-versions log*.

The following subsections describe how updates to each data structure are persisted in the log files.

5.3.1 Persisting mapping information

The version manager maintains the mapping information to store the directory names of each URL in the system. Each URL is mapped to a unique directory on the physical storage, but the entire URL string is not used as the directory name. The version manager uses a hash based approach [14] to generate a unique mapping name for the directory part of the URL as well as the file part of the URL. As an example, for a web page given by "http://www.uta.edu/current.html", the URL string is fragmented into two parts. The string corresponding to the file extension 'current.html' is considered as the file part and the initial part of the URL string 'www.uta.edu' is considered as the directory part of the URL. The directory part is mapped to a unique mapping for the directory 'D1'. The file part of the URL 'currrent.html' is mapped to 'F1'. The directory structure is a concatenation of directory mapping and file mapping. All versions of the page "http://www.uta.edu/current.html" are stored under the directory 'D1F1'. For URLs that do not contain any file extensions such as 'http://www.aa.com', only the directory mapping is maintained. In this case the directory name is 'D2'. Figure 5.1 shows examples of how URLs are mapped to a directory structure and the information regarding mapping is maintained in hashtables by the version manager.

The information corresponding to the mapping is maintained in two separate hashtables. One of them contains the directory mapping, and the other stores information regarding the file mapping. The updates to both the hashtables are logged to a

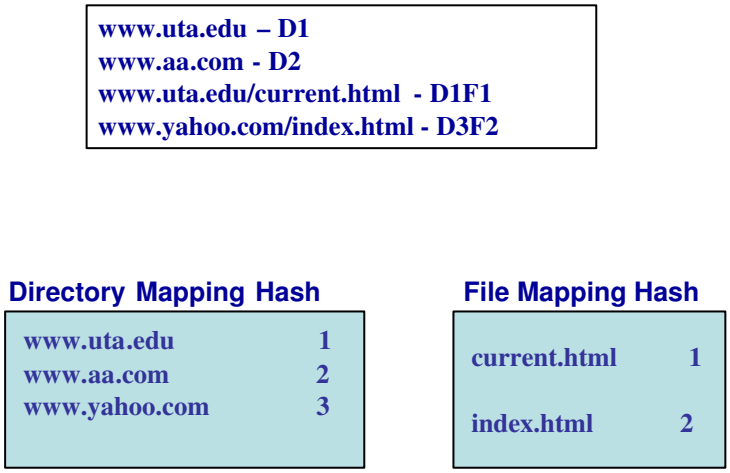


Figure 5.1 Mapping of URL names to a directory structure

single file called the mapping log. Before entries can be added to the hashtables, information is logged to mapping log. The entries in the mapping log are stored in a particular format, wherein each line in the log stores the information of URL and the corresponding mapping name separated by ##, this line is prefixed by an identifier which represents the data structure being updated.

Directory-mapping hashtable: The directory-mapping hashtable stores the directory part of the URL string as a key and the corresponding mapping name as the value. For the URL "http://www.uta.edu/current.html", the hashtable stores 'www.uta.edu' as the key and '1' as the mapping value. Before a new entry is added to this hashtable, the information regarding the key and the key value is logged with an identifier "[iD]". The identifier "[iD]" represents an insert to the directory-mapping hashtable. The format for logging information corresponding to the directory-mapping hashtable is as follows

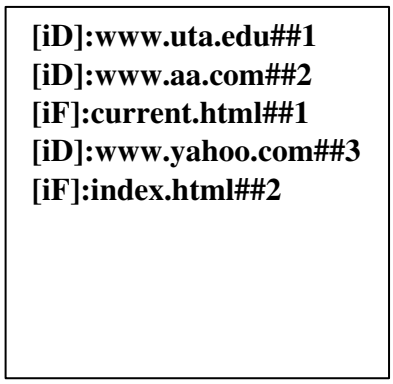


Figure 5.2 Mapping Log

- *[iD]:key##mappingValue*

File-mapping hashtable: File-mapping hashtable stores file part of the URL string as a key and the corresponding mapping name as the value. For the URL "http://www.uta.edu/curre the hashtable stores 'current.html' as the key and '1' as the mapping value. Information is logged in the similar fashion as directory-mapping hashtable but with identifier "[iF]" to represent an insert to file-mapping hashtable. Format for logging information for file-mapping hashtable is as follows

- *[iF]:key##mappingValue*

Figure 5.2 shows the mapping log for the URLs in Figure 5.1.

On recovery, when each line is read sequentially from the mapping log, the identifier is used to determine the action to be performed on the appropriate hashtable.

5.3.2 Persisting version information

The version manager [13] maintains three hashtables latest hash, best effort hash and fixed interval hash. Versions required for all the sentinels interested in a URL with best effort rule as its fetch type are stored in best effort hash. It has URL mapping as its key and list of version objects as value. Versions required for a sentinel which

has fixed interval rule as its fetch type are stored in fixed interval hash. It has sentinel id (sid) as key and list of version objects required for sentinel as value. Apart from the list of the versions, the value part of the hashtables store the information required for deleting versions of a URL. This information is based on the active sentinels in the system. Latest hash is a hashtable which maintains the latest version for every URL. It has URL mapping as key and stores the base version number and the details of the latest version for the corresponding URL as the value.

There can be version objects that are common to both the best effort hash and fixed interval hash as both the rules can be fired at the same time for the same URL. When a fetch rule is fired latest hash is checked for the URL. If it does not contain the URL, it indicates that the page is fetched for the first time. The page is stored and an entry is inserted into the latest hash and into the appropriate hashtable based on the fetch rule (best effort or fixed interval). Details of every version added to best effort or fixed interval hash are logged in the version log. Since the latest version of every URL keeps changing when a new version is fetched, the version inserted into a latest hash is not logged. During recovery, the hashtables fixed interval and best effort are built and the information of the latest hash is derived from these hashtables. The information required for deleting the version is not stored as this can be generated from the properties of sentinels that need to be restarted on recovery.

The logs are written in an append mode and details of each version added to best effort and fixed interval hashtables are appended in a new line. Identifiers are given to each line that is appended to the log. An identifier is important to represent the purpose of the information in the log.

Latest hash details: For each URL, only the base version information, which is important for the diff-patch approach is logged. The URL mapping and base version

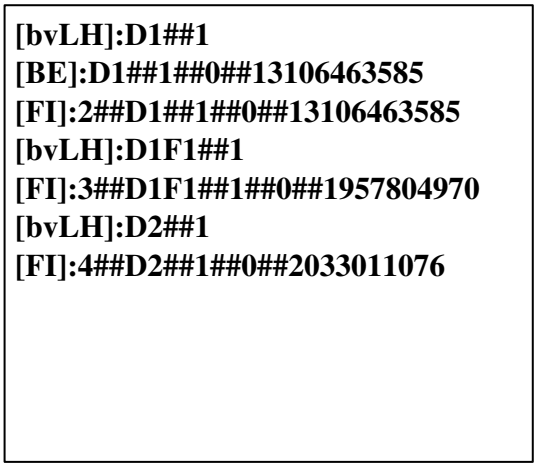


Figure 5.3 VersionLog

number are logged with an identifier "[bvLH]". The format for logging the updates to latest hash is as follows.

- *[bvLH]:URLMapping##VersionNumber*

Best effort hash details: Prior to an insertion of a version into the list of versions, the key value (URL Mapping) and the version object are logged with the identifier "[BE]". For each version, the version number, the last modified time and the checksum are persisted. The format for the updates to best effort hash is as follows.

- *[BE]:key##VersionNumber##LMT##Checksum*

Fixed interval hash details: The key value (sentinel id) and the version object are logged with the identifier "[FI]" before a version is inserted to the list of versions. The details of the version object such as the version number, URL mapping, the last modified time and the checksum are logged. The format for the updates to fixed interval hash is as follows.

- *[FI]:key##URLMapping##VersionNumber##LMT##Checksum*

The figure 5.3 shows the version log for the versions stored in the hashtables. The first entry in the figure 5.3 represents the log entry for the base version of a URL

with mapping D1. From the figure 5.2 it can be observed that the versions of URL http://www.uta.edu are stored in the directory D1. The second entry in the figure 5.3 represents the entry for a version of the URL page with mapping D1 that has been added to the best effort hashtable. On recovery, when each line is read sequentially from the version log, the identifier is used to determine the action to be performed on the appropriate hashtable.

5.3.3 Persisting information for deletion

During runtime there may be some versions that are added and others that are deleted from the hashtables. If a crash occurs after these versions are deleted from the hashtables, on recovery these versions should not be created and inserted into the hashtables in spite of the log entries corresponding to these versions in the version log. Additional information is required to avoid rebuilding versions during recovery that have been deleted prior to the crash.

The version manager maintains a deletion trigger for each URL based on the properties of the sentinels interested in that URL. Deletion of versions is triggered when the number of versions in the directory of a URL exceeds the deletion trigger. During the deletion process, the version manager determines the *oldest version*, V_O in the repository that is required for the sentinels active at that point. The best effort and fixed interval hashtables are scanned for versions older than V_O and are deleted. If the oldest version number V_O identified for a URL is stored in a log, it can be used during recovery to determine if a version needs to be reconstructed and added to the hashtable or not. The V_O maintained for a URL, corresponding to the URL mapping the version number is logged into deleted-versions log.

- *[dOld]:URLMapping##OldestVersionNumber*

During the recovery process, the mapping log is read first, and the hashtables for directory-mapping and file-mapping are reconstructed. The deleted-versions log is then read and a temporary hashtable with key as the URL mapping and value as the oldest version number is created. The version log is then read sequentially on line at a time, and based upon the identifiers specified in each line, the data structures are constructed. While adding a version of a URL to a hashtable (best effort or fixed interval), it is assured that the version number is not older than the oldest version (using the temporary hashtable of old versions) of each URL. As the version number for a version is given in the order of their fetch, oldness can be checked using version number. The versions in the log are added to the hashtables only if their version number is greater than or equal to the oldest version number of a particular URL. Also, while building the best effort and fixed interval hashtables, the latest hash is rebuilt such that it stores the base version and the latest version of each URL. Logs are read sequentially line by line from begining to end. This is to rebuild the data structures in the same way they were actually constructed.

5.4 Starting sentinels for monitoring

Information regarding sentinels is persisted in the knowledge base of the system. The knowledge base in WebVigiL consists of relational tables in a database that store the information such as the page of interest of a sentinel, the sentinel start time and end time etc. During the recovery process, all sentinels that need to be restarted are identified using the knowledge base. When the system crashes, the failure in the system has to be detected and the system has to be restored to a consistent state before starting the sentinels. As there can be latency between the point of failure and time to recover, the time point of recovery is considered as the reference for starting the sentinels. The end time of a sentinel is compared with the time of recovery. If the time point of recovery is close to the end time of a sentinel such that their difference is less than the fetch frequency

of the sentinel, there might not be any pages fetched and thus no change detection. Such sentinels need not be restarted during recovery. Sentinels that have to be restarted should have their end time scheduled after the time point of recovery and should have at least one fetch cycle from the time of recovery till the sentinel ends. ECA rules [11, 12] have to be created for the sentinels that have been identified to restart.

5.4.1 ECA rules for restarting the sentinels

The events and rules required for change monitoring and notification have to be generated for all sentinels that need to be restarted. For each sentinel, the activation rules are created based on the sentinel start and end time. The start time and end time of the sentinel are persisted as absolute values in the knowledge base. If the start time of a sentinel is prior to the recovery time, it implies that the sentinel was active at the time of crash. For such sentinels rules are created with the time of recovery as the start time. The fetch rules required for fetch and notification are created based on the fetch interval and other appropriate information contained in the knowledge base. Once the rules are created the system performs the necessary operations.

5.5 Server-based WebVigiL

The WebVigiL system is capable of being run as a server, when the server starts, it initializes various modules required for activation, change detection and notification. The ECA rule generator required for accomplishing active capability is also initialized. The server is then ready to accept requests from users. The user interface provided for placing a sentinel is run on a webserver. The webserver and the WebVigiL server can be on the same machine or different machines. When a user places a sentinel, it is validated and sent to the WebVigiL server. The server accepts the sentinel request and informs the respective module to process the request. The events and rules required for fetching and

notification for the sentinel are generated using the ECA rule generator. With such a server, the dependency with the web server that runs the user interface is removed. The server can be requested to perform a graceful shutdown to close down the monitoring functionality. The section 5.5.1 discusses the process of graceful shutdown in WebVigiL.

5.5.1 Graceful shutdown for WebVigiL

A graceful shutdown of the system ensures a proper close down of the functionality of each module in the system. A request to the server can be sent for a graceful shutdown. During the process of shutdown, each module completes the process in progress and closes itself. For a proper close down, the modules have to be closed in a particular sequence. The server ensures that the sentinel requests accepted until shutdown are persisted into the database. The server then stops listening to user requests, this prevents the system to accept any new requests. The activation module responsible for starting the rules is closed after completing the process in progress. The events and rules registered with ECA rule generator are halted. Internally ECA rule generator runs the events as threads, when the object of ECA rule generator is destroyed and the main program (WebVigiL system) ends these threads are closed. The change detection module closes after finishing its processing. Finally, the notification module notifies the changes (if any) to users and closes down. When the WebVigiL system is down, it will not accept any requests from the user. User specification module will not allow users to request for any sentinel when the system is down. User is prompted with a message of the system being down when he/she tries to access the interface for placing a sentinel.

5.6 Failure detection and recovery

During the downtime of the WebVigiL system, monitoring for sentinels in the system is stopped and no new requests are accepted. To minimize the downtime of the

system and to ensure swift recovery of the system, a failure detection mechanism has been introduced. Failure detection is achieved by scheduling a failure detector periodically. The current WebVigiL system is developed on Linux using the Java runtime. The crontab facility (a scheduler for Unix-based systems) is used to run a failure detector program. A file specifying the periodicity and the task to be performed is given to crontab [23], and it performs the task according to the specified periodicity. For WebVigiL, the periodicity is specified as two minutes and the task involves running a script file to monitor the processes corresponding to the WebVigiL server. If there are no processes running for WebVigiL, a failure is detected and the WebVigil server is restarted.

The recovery module maintains a flag called recovery lock in the system. This is a configurable parameter and is set to *false*. The updates to data structures are logged into the corresponding files only if the recovery lock is disabled (when it is false). When WebVigiL starts, the modules are initialized, recovery lock is set to default value (from configuration file) and the control is given to the recovery module. During recovery, the recovery lock is enabled (flag is set to *true*) before reading the log files for reconstructing the required data structures. Recovery lock is enabled to avoid logging the updates to data structures during recovery. Once the data structures are built, log refreshing (discussed in section 5.6.1) is performed over version log and deleted-versions log. The recovery lock is disabled (flag is set to *false*) once the runtime data structures are rebuilt. Sentinels that need to be started are identified, and the required events and rules are created to process these sentinels. As the sentinels are processed, the required rules are generated, the grouping information and the graphs required for change detection are created. The recovery module hands the control back to WebVigiL server which will be listening to user requests. When a graceful shutdown is desired, the failure detection program is terminated before closing down all the modules. Currently the

failure dectection program scheduled with crontab is removed manually from the cronfile. Once the system is started, the failure detection program is added to cronfile.

5.6.1 Log refreshing

During runtime, versions details are inserted and deleted from the hashtables best effort hash and fixed interval hash. At the time of recovery, there may be entries in the log for versions that are deleted from these hashtables. Once the hashtables are reconstructed from the log such entries can be removed from the version log. Log refreshing is done on the version log to remove such entries and refresh the version log to represent the exact information in the hashtables. Required information from the hashtables best effort hash, fixed interval hash and latest hash are written to a temporary log file. The current version log is deleted and the temporary log file is saved as version log. Only after the version log is refreshed, are the entries in the deleted-versions log deleted. In the current system, as the entries are never deleted from directory-mapping hashtable and file-mapping hashtable, log refreshing is not performed over mapping log.

5.7 Summary

The current WebVigiL system is provided with server capability and runs on a machine listening to and accepting requests from users. The server can close down gracefully without losing runtime data. The stability of the system has been addressed by introducing a recovery module. The various modules of the system are given additional functionality to log the details of the runtime data structures that contain crucial information. The recovery module uses these logs to stabilize the system after a crash. A failure detection mechanism has also been introduced into the system to ensure the availability of the system at all times.

CHAPTER 6

EVALUATION OF SERVER-SIDE FETCH IN WEBVIGIL

Web monitoring tools need to fetch versions of pages to detect changes between versions. Changes can be detected only when relatively different versions of a page are compared. The fetch module of WebVigiL fetches the versions of the URL page requested by the user. This chapter discusses the current fetch mechanisms of WebVigiL, how it can be improved with respect to the number of fetches and the data transfer that occurs. A server-side fetch mechanism is proposed to evaluate the difference between server-side fetch versus WebVigiL fetch. It also compares the cost effectiveness of the server-side fetch mechanism with respect to the existing fetch module of WebVigiL.

6.1 Current fetch module

The fetch module of WebVigiL is responsible for fetching the versions of a page required for a sentinel. In WebVigiL architecture, shown in the figure 2.1 (in the chapter 2), the fetch module resides in the WebVigiL middleware along with other modules. Fetch module uses the *fixed-interval rule* to fetch pages when user specifies a particular fetch interval. When the user does not specify a fetch interval, the fetch module uses the *best effort rule* [4] to tune to the change frequency of the requested URL page. A detailed description of the fetch module is given in chapter 2. The fetch module ensures that only fresh versions of a page are retrieved and stored in the repository. Freshness is determined on the basis of page properties of a web page.

6.1.1 Page properties

The fetch module uses the page properties or metadata to determine the freshness of a web page. Metadata used, for determining the freshness of a page, are: last modified time stamp (LMT) and page size or checksum of the page. Depending upon the nature of the page being monitored, the complete set or subset of the metadata is used to evaluate the change. When the fetch module connects to the URL page, LMT and page size can be obtained from the HTTP header of the page. It compares these values with the latest version of the page available in the repository. If the values differ, it is considered to be a change and the contents of the page are retrieved using the 'HTTP GET' request. Web pages may or may not have a last modified time stamp (LMT) associated with them. For pages that are dynamically generated on the web, the last modified time stamp is not provided. For such pages, checksum of the page contents is calculated and compared with the checksum of the previous page.

Figure 6.1 shows the current setup of the fetch module. When the LMT of a URL page has to be obtained, a connection is established to the URL page. Another connection is established to the page if the page content has to be retrieved. Two connections are required to check the LMT and retrieve the page. However, even if the page is not retrieved (i.e., when the LMT remains same as the previous version), one connection is required to get the LMT. But for pages where checksum is compared, two connections are required to decide whether the page should be stored or not. One connection to identify that LMT is not available and one to retrieve the content (to compute checksum) of the page. Such cases result in retrieving the contents of the page for computing checksum, even when the pages are not stored.

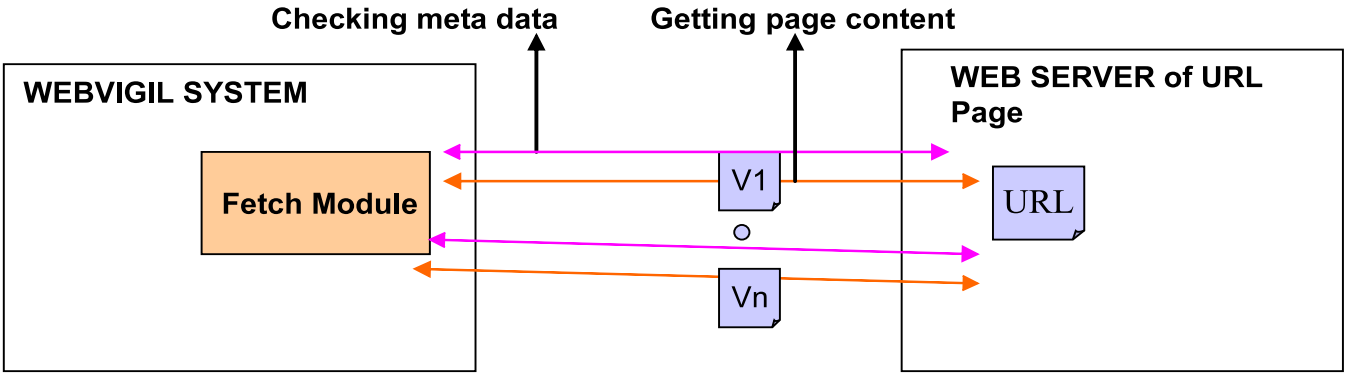


Figure 6.1 Current fetch module at WebVigiL system

6.1.2 Issues with the current approach

WebVigiL system allows users to specify a time interval for fetching. The fetch interval provided by the user may not be the exact interval with which the page changes. However, the fetch rule created for this sentinel tries to fetch a new version periodically. If the page requested by the user does not change frequently, the fetch module ends up connecting to the web page only to get the metadata of the page. Consider a scenario where a user places a request to monitor 'links and images' on the web page *http://www.hotwire.com*, every one hour for 10 days. The sentinel specified for this scenario is as follows:

Create Sentinel s1 using http://www.hotwire.com

Monitor links AND images

Fetch every 1 hour

From 10/01/04 To 10/10/04

Notify By email bob@cse.uta.edu Every 4 days

Compare pairwise

When this request is registered with WebVigiL, the fetch module starts the rules to fetch the page every one hour (using interval based rules). Every one hour, the fetch rule

connects to the web page to get the metadata. The page specified here does not provide the last modified time in the metadata, so another connection is established to retrieve the contents and compute the checksum. Every hour there would two connections from fetch rule (at WebVigiL system) to the URL page. In one day, there would be totally 48 times that fetch rule connects to the URL page. Here if the page changes once per day, the page will be stored in the repository only once. In such scenarios, it can be observed that URL page is connected 48 times but the page is fetched only once. Over a period of 10 days, there will be 480 connections but the page is actually fetched only 10 times. These connections add to the network traffic and data transfer.

Data is dynamically published or rendered onto most of the news websites and e-commerce websites. For such pages, LMT values are not maintained in the header information. This increases the probability of dealing with pages where checksum is required as the criteria for deciding the freshness. The connection costs in the case of dynamic pages or pages where LMT information is not available are high as each attempt to fetch a page costs two connections. There is a need for an approach to reduce such connections form the WebVigiL system while fetching pages. Server-side approach is proposed to reduce such connections and the data transfer.

6.2 Server-side approach

It can be observed from figure 6.1 that the fetch module at WebVigiL system follows the pull paradigm. It pulls the data from web pages and provides it to the modules of WebVigiL. In pull paradigm, even for the best case, where a page is fetched in every poll, there are two connections to the page (one for checking the metadata and one for retrieving the content). Connections to the web servers cannot be avoided if the pages have to be retrieved using the pull paradigm. However, the connections to get the metadata of the page can be reduced. Typically, instead of polling the pages for a

change and then retrieving, the fetch module can use a push mechanism by maintaining a system on the web server of the web page.

Several researchers have investigated the push based technologies [24]. The underlying assumption in most of the push mechanisms is that the server (webserver of URL page) provides additional capability of pushing the updates. It has a push system which maintains the profiles of users interested in updates to web pages. The server provides the updates to these push systems which then propagate them to interested users. For instance, American Airlines [25] uses a mailing list as a push system to send the NetSaver Fare Alerts to all the subscribers on list. Such systems notify the updates to users directly. Customized change detection and selective notification cannot be achieved with this approach.

A pull-based push system having minimal modules of WebVigiL on the web server is introduced to achieve the push mechanism. Such pull based push system saves the connections from WebVigiL system to periodically monitor for metadata changes. This will be referred as "server-side fetch module " in further discussions. If a push system with customized change detection and selective notification is desired, then the web server needs to have the change detection module running on it. The change detection graphs required for performing change detection should be built on the server-side, the required pages should be stored in a temporary repository and the information about the repository should be maintained by version controller module. This adds a considerable amount of overhead on the web server of the URL page.

In server-side approach, only the modules required for monitoring the metadata of a page, exist at the web server. These modules will trigger light weight processes in the form of LED rules for monitoring. So it is ensured that the web server is not loaded with too many processes of WebVigiL. The server-side fetch module polls the pages from the web server of a URL according to the specifications of the user. If a change in the

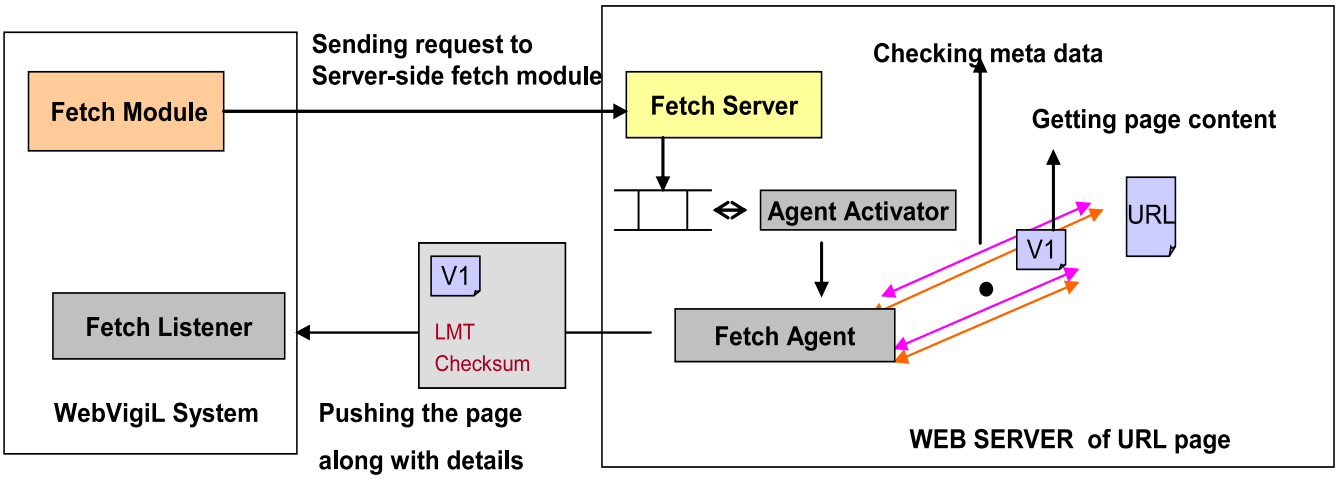


Figure 6.2 Server-side fetch module

metadata is found, the page is pushed to the fetch module at WebVigiL system. The pages are stored and the change detection is performed at the WebVigiL system. The modules required for storing the pages, change detection and notification will be running at the WebVigiL system.

6.2.1 Server-side fetch module

This section discusses the design of the server-side fetch module. Monitoring of a page should begin and end according to the sentinel specifications. The changed pages(in terms of metadata) should be pushed to the WebVigiL server. At the WebVigiL system, there should be some mechanism to receive these pages, store them in the repository and inform the change detection module.

The figure 6.2 shows the overview of the functionality of Server-side fetch module. If the server-side fetch module exists for URL page *www.xyz.com*, it resides at the web server of the web page. When a user creates a sentinel for URL page www.xyz.com, the events and rules required for the sentinel are created at WebVigiL server. When the sentinel starts, the fetching job is delegated to the server-side fetch module. The

information of the URL page, URL mapping, fetch interval and fetch type are sent to server-side fetch module. *Fetch agents* monitor the pages at the web server and inform the WebVigiL system. These pages are received by the *fetch listener* at the WebVigiL system.

The server-side fetch module consists of a *fetch server* and an *agent activator*. *Fetch server* is a server program that runs on the web server of the URL page which listens for requests from clients. When the sentinel starts, the details of request are sent to *fetch server* which propagates the request to *agent activator*. *Agent activator* starts the *fetch agents* to monitor the requested page for changes in the metadata. Whenever a change in the metadata is observed, the fetch agent pushes the contents of the page to the fetch module at the WebVigiL system. At the WebVigiL system, the *fetch listener* receives the pages, stores the pages in repository and informs the change detection module regarding the new version.

Agent activator and fetch agent: Agent activator is a part of the server-side fetch module to process requests from the clients. It is implemented as a thread which waits on a buffer. When requests are placed in the buffer, *agent activator* reads the request and accordingly starts the *fetch agent*. *Fetch agents* are activated using ECA paradigm to perform monitoring of the metadata of the page, they reside at the server-side fetch module.

The ECA paradigm is used to start the events and rules (best effort and fixed interval) for fetching as per the request. Minimum information required to determine the freshness of a page that is fetched is maintained. When a change in the metadata of a page is observed, the page content with additional information like the URL page name and the details of the version (viz., the version name, version number, LMT and checksum value) are pushed to the WebVigiL system. When the sentinel ends, the fetch server is requested to end the monitoring of the specified page. Fetch server informs the

fetch agent to terminate the events and rule for fetching and close down the monitoring process. The number of rules generated at server-side end depends upon the number of requests.

6.2.2 Issues with server-side fetching

Server-side fetching involves the issues of starting the processes required for monitoring on the web server side. Fetch events and rules are created at the web server of the URL page. This should be handled such that they do not overload the web server.

As the number of requests increase, the rules at server-side increase. The overhead due to these rules and the push connections to the WebVigiL system may not be acceptable to the web servers. To balance the load, a load factor is provided at the server-side fetch module. The load factor is a configurable parameter which represents the number of requests that can be processed from the server-side. When the number of requests being processed at server-side is equal to the load factor, additional requests will not be processed. The fetch server redirects the request to perform fetching from the fetch module at the WebVigiL system.

With the server-side approach, there will be no connections to the web server for metadata from WebVigiL system. The pages will be pushed from fetch agents whenever the page changes.

6.3 Effectiveness of server-side fetch module

The following parameters are considered to analyze the cost effectiveness of server-side fetch.

- N_M = Number of polls/look-ups for metadata of a page
- N_P = Number of times the page has been actually fetched
- D_M = Amount of data transferred as metadata

- D_P = Amount of data transferred while retrieving content of the page
- D_S = Amount of data transferred when data is pushed from server-side fetch (D_P + d where d=size of additional information for a version)
- D_T = Total amount of data transferred
- N_C = Total number of connections to web pages

These parameters are used for analyzing the cost effectiveness in the following scenarios.

1. *Fetch module at WebVigiL system when LMT of URL page is available:* In such cases, the fetch module checks the LMT to determine the freshness of a page, and fetches the page accordingly. Each attempt to check the metadata of a page costs a connection to obtain header information. If a change is observed, another connection is made to obtain page content.

   Total number of connections, $N_C = N_P + N_M$

   Total data transfer, $D_T = N_M * D_M + N_P * D_P$

2. *Fetch module at WebVigiL system when LMT of URL page is not available:* In such cases, the fetch module uses the checksum to determine the freshness of the page and fetches it accordingly. Each attempt to fetch a page costs two connections.

   Total number of connections, $N_C = 2 * N_M$

   Total data transfer, $D_T = N_M * [D_M + D_P]$

3. *Server-side fetch module exists:* In such cases, the server-side fetch module pushes the pages only when a change is observed in the metadata. Connections to WebVigiL server are established only when a page has to be pushed. Content of a page is pushed along with the information of version.

   Total number of connections to WebVigiL, $N_C = N_P$

   Total data transfer, $D_T = N_P * D_S$

In this case, the number of connections established to WebVigiL system and the data transfer between WebVigiL system and web server remain the same for pages with and without LMT.

6.3.1 Experimental Setup

To observe the effect of the parameters discussed, experiments have been conducted. In the experimental setup, a web page *testPage.html* at cise.ufl.edu server (a machine at the University of Florida, Gainesville) was considered. Server-side fetch module was placed on this server.

A sentinel was placed at WebVigiL requesting for the page. The following are the specifications of the sentinel.

Create Sentinel s1 Using http://www.cise.ufl.edu/ sharma/testPage.html

Monitor all links AND all images

Fetch every 1 hour

From 10/01/04 To 10/10/04

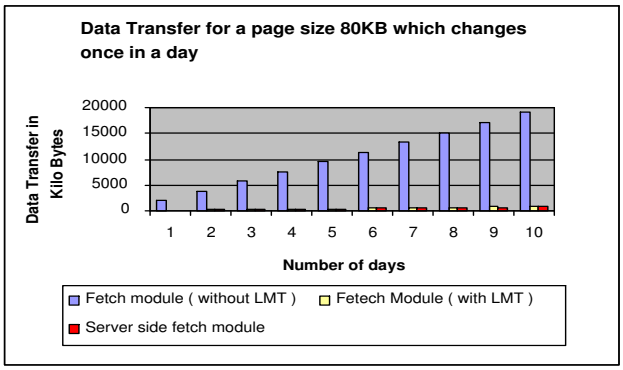
Notify By email bob@cse.uta.edu Immediate

Compare pairwise

When the page is accessed from the WebVigiL system running at itlab.uta.edu server (machine at The University of Texas at Arlington), the value of D_M was observed to be 183 bytes and the page size D_P is 80949 bytes. With server-side fetch, it is assumed that the page will be sent to the fetch module at the WebVigiL system with some additional information and so the value of D_S is 80999 bytes.

The data transfer and the number of connections from WebVigiL was compared for the following scenarios

1. *Scenario 1:* Fetch module at WebVigiL system when LMT of URL page is available



$N_M = 240$, $N_P = 10$

	N_C	D_T (KB)
Fetch Module (without LMT)	480	19015
Fetch Module (with LMT)	250	833
Server-side fetch module	10	791

Figure 6.3 Data transfer and connections

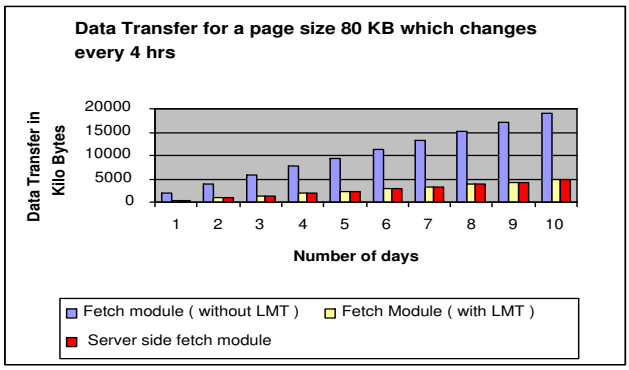
2. *Scenario 2:* Fetch module at WebVigiL system when LMT of URL page is not available
3. *Scenario 3:* Server-side fetch module

The three scenarios have been compared assuming different change frequencies for the URL page http://www.cise.ufl.edu/ sharma/testPage.html.

When the page changes once in a day: For the above sentinel, fetch rules check the metadata of URL page every one hour. Connections are established for the page to obtain LMT value and the actual page is fetched only when the LMT changes (once a day in this case).

- Number of times the LMT is checked is 24 times in one day. Therefore, number of polls in 10 days will be $N_M = 24 * 10 = 240$
- But as the page change only once in a day, the number of times page will be actually fetched will be N_P=10

Figure 6.3 shows the data transfer and the connections that will be established for this case. It can be observed that the number of connections is least with the server-side fetch module (Scenario 3). The data transfer involved will be drastically high for scenario 2 when compared to data transfer for scenario 1 and 3. This is because in scenario 2, the



$N_M = 240$, $N_P = 60$

	N_C	D_T (KB)
Fetch Module (without LMT)	480	19015
Fetch Module (with LMT)	300	4786
Serverside fetch module	60	4746

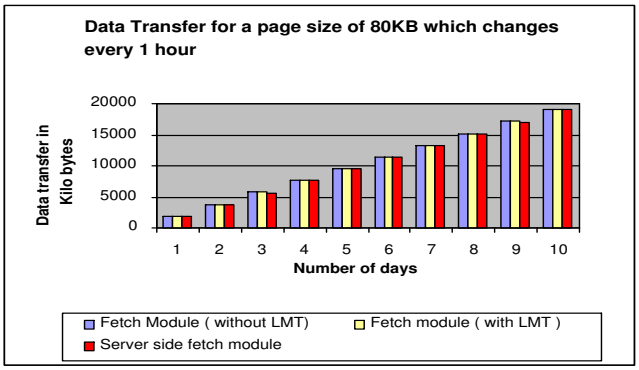
Figure 6.4 Data transfer and connections

data in terms of metadata and page content are transferred in every fetch even though the page does not change. The data transfer for the scenario 1 is slightly more than data transfer for scenario 3. This is due to data transfer of metadata information to check the LMT value every hour.

When the page changes every four hours : For the above sentinel, the fetch rules check the metadata of URL page every one hour. As the page changes every four hours, it is actually fetched only 6 times in a day.

- Number of times LMT is checked is 24 times in one day. Therefore, number of polls in 10 days will be $N_M = 24 * 10 = 240$
- The number of times page will be actually fetched in a day is 6. Therefore, for 10 days N_P=60

Figure 6.4 shows the data transfer and the connections that will be established for this case. It can be observed that the number of connections is least with the server-side fetch module (scenario 3). This is because, with the server-side fetch, the page is pushed only when it is changed. The connections to WebVigiL system are established only to push the changed pages and the connections and data transfer involved with checking the metadata is avoided.



$N_M = 240$, $N_P = 240$

	N_C	D_T (KB)
Fetch Module (without LMT)	480	19015
Fetch Module (with LMT)	480	19015
Serverside fetch module	240	18984

Figure 6.5 Data transfer and connections

When the page changes every 1 hour : For the above sentinel, fetch rules check for the metadata of URL page every one hour. As the page changes every one hour, the contents of the page have to be retrieved every hour. With server-side fetch, the contents have to be pushed every hour.

- Number of times LMT is checked is 24 times in one day. Therefore, number of polls in 10 days will be $N_M = 24 * 10 = 240$
- Page is fetched every time it is polled(checked for metadata). Therefore, for 10 days N_P=N_M=240

Figure 6.5 shows the data transfer and the connections that will be established for this case. It can be observed that the number of connections and data transfer is same for scenarios 1 and 2. This is because the page contents are retrieved in every poll. The data transfer in scenario 3 is almost same as the other two scenarios. However, even in this case the number of connections are less for scenario 3.

6.3.2 Conclusions and Summary

From the above analysis it can be concluded that having a server-side fetch will reduce the overhead of data transfers to a large extent when pages without LMT are

involved. Even for pages with LMT, as we observed, the data transfer overhead is present, but can be considered negligible. The gain for such pages is in terms of the connections established from the WebVigiL system. The connections for checking the LMT value can be avoided when the server-side fetch module is used. The complexity in having the server-side fetch module is the feasibility of running it on the web server machine. The work in this thesis presents the cases where such a module is feasible. We observed that even when the pages change at every fetch, there is gain in terms of connections established by using the server-side fetch module. The number of fetch requests being processed at the server-side can be controlled by using a load factor to keep track of the number of fetch requests. For cases where the number of requests exceed the load factor, fetching can be performed from fetch module at WebVigiL system.

CHAPTER 7

RELATED WORK

This chapter discusses some of the related works in the areas of version management (using the diff approach), recovery, and fetching which form the topics of this thesis. Also the related work in the field of web monitoring is discussed in the last section.

7.1 Version management with diff-patch approach

This section provides a brief overview of the existing systems that deal with maintaining versions of a document.

SCCS and RCS The popular tools/systems used for version management are RCS and SCCS. RCS [21] maintains the most recent version of a file in its entirety (complete contents) and uses edit scripts for maintaining the *diff files*. **Diff files** store the changes in older versions of the document with respect to the most recent version. CVSIt [18] is a versioning system based on RCS. Both RCS and CVS considers most recently used version as the base version. When an updated file is added, the existing base version is used to regenerate the original contents of older versions (stored as diff files). The newly added version is considered as base version and the diff files for all older versions are created with respect to the new base version. It would be an overhead to use these tools to WebVigiL as the above steps have to be carried out every time a new version is fetched.

SCCS [19] is a time-based scheme where a unique version identifier is given to the object of interest. Here the first version is maintained in its entirety and all other versions are maintained as changes with respect to first version. RCS internally

computes the changes between two pages in the same way as the diff utility. The handle to individual diff files is not provided using RCS or SCCS. The approach that is applied for WebVigiL is to use the diff utility [20] of Unix-based systems. For windows systems, the diff utility can be supported if Cygwin [26] is installed.

Utilities to compute difference between two files There are different utilities provided by Unix for computing the difference between two files apart from diff utility. The *cmp* [20] utility compares two files of any type and writes the results to the standard output. The result is the byte and line number at which the first difference occurred. But as the result does not actually give the difference in terms of the changed content, there is no mechanism to get the contents of the original files from the result of the difference. *sdiff* is a utility used to show the difference between two files. It presents the difference between two files and produces the difference in a side-by-side format (presenting the lines that are different in both files). The diff files generated may have same size as the original sizes and may not reduce the storage space. It is not applied to WebVigiL as the idea of storing the diff instead of the original file is to reduce the storage size of the repository. And as this is a tool to represent the changes, no mechanism is provided to regenerate the original contents from the result of sdiff.

Xyleme: It [27] uses a change-centric approach as opposed to data-centric approach which is more frequently adopted for database versioning. XyDiff, Xyleme Zone Servers versioning feature, computes changes between document versions. Changes to documents are represented as XML delta files and can thus be stored, indexed and queried within the Xyleme Zone Server. These delta files are also used to reconstruct previous document versions. Given two consecutive versions of a page, using a very efficient diff algorithm the deltas are computed. Deltas are in XML format and contain the additional information required to regenerate the contents of the

actual file. Persistent identifiers are essential to represent and control changes. The authors believe that using deltas can be very useful to version results of continuous queries. In WebVigiL, the complete version of the page has to be stored to be given to the change detection module to compute customized changes. Even if the deltas are computed, there should be a mechanism to generate the original versions from the deltas. Generating the original contents from the deltas involves some computation as the files need to be parsed to understand the information represented in XML trees. This defeats the purpose of storing deltas in the repository.

7.2 Fetching

In WebVigiL, fetch module residing on the WebVigiL system will poll the web servers of different web pages to obtain the data. To reduce the wasteful connections server-side fetch module which uses an intelligent pull, push system is proposed. Some related research in terms of balancing the use of push and pull are presented here.

Balancing Push and Pull for monitoring changes An adaptive Push-Pull mechanism is discussed in [1]. In this work, a proxy server is involved between the user and his/her page of interest. Proxy server maintains the profile of user's interest. The proxy server monitors the changes to page of interest by applying push mechanism or pull mechanism. User provides the page of interest and the temporal coherency requirement(tcr). Tcr denotes the maximum permissible deviation of the cached value from the value at the server. It can be specified in units of time (e.g., the version of the page should never be out-of-sync by more than 10 minutes). For pull mechanism the proxy decides the fetch interval (when a page has to be fetched) based on rate of change of the data and the tcr value specified by the user. For push mechanism, the proxy registers with the web server (of the web page) identifying the data of interest and the associated tcr. When the data changes, the

server sends the changed data to proxy server which then propagates to interested users.

Pap and Pop are two techniques for combining push and pull. In Pap, proxy initially follows push mechanism. If the push mechanism stops, pull comes into picture. For Pop, the proxy adaptively chooses between push and pull. This is based on the behavior of changes in page. If the page is rapidly changing, push is used otherwise pull is used. If the server does not have enough resources then it shifts some push based requests which have a greater tcr value to pull based requests.

The *temporal coherency requirements*, computational overheads, space overheads and failure resiliency are considered as the parameters to decide between push and pull to be used for a given page. Push mechanism in this approach depends upon the web server. If the web server of the page does not provide such capability then the push mechanism cannot be applied. To achieve selective notification using push mechanism, the state information of users (interests of users) is maintained at the webs server. Server-side fetch module does not depend upon the web server to provide any information about the updates. It uses the pull mechanism to identify the updates. It maintains minimum information required to identify a change in metadata. Change computation and notification are carried out at WebVigiL system.

Balancing Push and Pull for disseminating data Research objectives for balancing push and pull for disseminating data are different. In these systems, it is assumed that the push and pull capability are given by the web server which maintains pages. [28] gives approaches to balance between push and pull. In this model, an application process of the web server resides on the user workstation. Monitoring is performed using this application process if pull mechanism is desired. The application process generates pull requests for required data (as per user re-

quirements) and propagates the changes to clients. For achieving push mechanism, the server sends updated data to this application process. Push is scheduled by the server as it can either be in a periodic fashion (if the user sets a time interval) or as and when the data changes. Here, the parameters that have been considered for balancing the push and pull mechanism [28] are mainly to balance the load at the web server. Load in terms of processing the user requests (when they poll the page in a regular fashion) and in pushing the data to various clients (registered with the push system). Summary of this work is that push based approach is better to handle situations when there are too many users polling server for different pages. This is because, pull is a request-response based system and when there are too many requests at the server all of them may not respond because of the overhead of network traffic. But when there is no contention at the server, pull mechanism works well. For varying loads they proposed an algorithm Interleaved Push and Pull (IPP) which mixes both push and pull based on the parameters like bandwidth usage, response time at the client. The thresholds for bandwidth usage are predetermined.

In all push based systems, it is assumed that server provides the updates to pages. Push or pull are decided by the load on the systems or servers involved. Server-side fetch module presented in this thesis is like pull based push system. It resides on the web server of the web page, pulls the pages and pushes them to WebVigiL system. The process of polling is to identify the fresh pages that need to be pushed to WebVigiL for change detection as it is assumed that the server does not provide any information about the updates. Minimum information required for fetching is maintained on the web server of the pages. The emphasis here is to reduce connections from WebVigiL system.

7.3 Recovery

A DBMS has a recovery manager that maintains relevant information during normal execution of transactions. A log of all the modifications to the database is saved on stable storage. The log enables recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. Recovery manager ensures atomicity by undoing the actions of uncommitted transactions and ensures durability by making sure that all actions of committed transactions are persistent. Initial approach to database recovery was an UNDO/REDO approach. ARIES [29] is a mechanism which replaced this approach.

Algorithm for Recovery and Isolation using Event Semantics (ARIES) [29, 30] is a Write Ahead Log (WAL) based recovery mechanism. It supports fine granularity locking and partial rollbacks. Aries uses log files to record the actions that cause changes to data objects. It records all transaction into a log which is considered as a sequentially growing file. This log is critical for ensuring a transactions committed actions are reflected in the database despite various types of failures and that its uncommitted actions are undone. Aries supports page-oriented redo and logical undo.

The WAL protocol asserts that the log records representing changes to some data must be saved on stable storage before changing the actual data in nonvolatile storage. The system is not allowed to write an updated page to the nonvolatile storage version of the database until at least the undo portions of the log records have been written to stable storage. To enable the enforcement of this protocol, systems using the WAL method of recovery store in every page, the LSN of the log record that describes the most recent update performed on that page. LSN is a unique log sequence number assigned to the record when that record is appended to the log. LSNs are assigned in ascending sequence.

Unlike DBMS, WebVigiL has no transactions. When WebVigiL server crashes, the runtime data structures maintained by version manager module are lost. WAL is used to persist the updates to these data structures. As the requirement here is to persist the updates to data structures, the concepts for undo, redo operations have not been applied. During recovery, the log files are read sequentially and the data structures are rebuilt.

7.4 Buffer management and replacement policies

In Operating systems and database management systems, the goal of buffer management [31] is to provide access to more data pages with minimum accesses to main memory. A buffer is maintained with pages that are once accessed from main memory and replacement policies are applied to remove pages that are no longer needed. In Operating systems there is no information of the pages that are going to be required in the future so the replacement policies depend upon the history of usage. The recency factor and the usage are considered to identify a victim object. Traditional replacement policies are LRU, MRU, LFU, MFU, Clock and FIFO. In database management systems, the pages that will be accessed can be predicted. This prediction can be used to maintain the pages in the buffer. Pre-fetching is a concept that is applied in IBM DB2, Oracle 8 and Microsoft SQL, where the buffer manager anticipates the pages that will be accessed and fetches from memory in advance to its requirement.

For WebVigiL, the requirement of the buffer is to maintain the runtime objects that are going to be reused. Buffer is better utilized if objects that have better reusability are retained. The traditional polices cannot always choose the best victim object. Pre-fetching cannot be applied when runtime objects of URL pages have to be stored in the buffer. So an approach which gives priority to reusability factor is proposed.

7.5 Change detection

WebCQ It [32] 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 continuous queries. WebCQ is designed to discover and detect changes to web pages and to provide a personalized notification of the changes to the users. Users' update to monitoring requests are modeled as continuous queries on the web. WebCQ change detection robot is responsible for discovering and detecting changes to web pages. WebCQ provides a trigger condition parameter by which the user can specify how frequently the change detection robot should be fired to check if any interesting changes have taken place and how soon the notification should be sent. WebCQ does not archive versions of documents, but it uses object cache update, wherein it stores the object in the old version that has changed in the new version. As WebVigiL provides the option of comparing the documents with not only the previous version but also a relatively older version.

RCS RSS [33] stands for 'Rich Site Summary', it is an XML-based specification format that is used to syndicate news sites, personal web logs etc. Web pages featuring RSS publish a file which contains the required information in the XML format. This file is updated whenever there is a change to the published information on that page. To keep track of these changes, a reader application is required which periodically checks the published RSS file and notifies of changes. RSS only provides a basic framework for change presentation and the complexity and usability of the published changes depend on the RSS reader. Most of the freely available RSS readers provide the facility to only highlight the new information that is published. There are some commercial RSS readers [34] which support some additional features like monitoring a feed for specific keywords or phrases. RSS is limited in scope, not all web pages give the RSS feeds. The information that is received through

RSS has to be read manually or some other parser of interpreter should be used to convert the information to user readable format. WebVigiL presents the changes using *only change* or *dual frame approach* as per user specification.

Alerts Nowadays the alert functionality provided by 'Yahoo!' and 'Google'to track changes happening to web pages are gaining popularity. 'Yahoo!' provides a web alert feature [35] . Using this feature, users specify their criteria based on a directory structure which provides pre-configured categories such as auctions, autos, breaking news, weather, etc. These categories have pre-classified web pages and the user cannot specify a particular web page to monitor. This greatly limits the functionality of the service. Also, there is no feature to specify the required change type at a finer level of granularity like links, images, etc. Also, there is no feature to specify how frequently the pages should be checked for changes. The online search engine 'Google' also provides a web alert service [36] using which the user can specify a set of keywords. Whenever the search results returned by the search engine changes for the specified keywords, an e-mail notification is sent. The search results to monitor can be restricted to either the search on the entire web content or the search on popular news websites. Monitoring can be done with a frequency of 1 day or 1 week or as it happens by using a proprietary algorithm. This also lacks the functionality to monitor specific web pages at a finer level of granularity provided by WebVigiL. Even though these services monitor multiple web pages, the user cannot specify what web pages to monitor.

Wysigot Wysigot [37] is a client-based stand-alone application to monitor HTML pages. This application has to be installed and always running on the client machine, which limits its applicability. It provides three options to specify the fetching of a page: automatic, periodic and manual. In the automatic mode, it uses a proprietary algorithm to fetch pages according to the history of observed changes; in

the periodic mode it fetches pages according to the user-specified time interval, and in manual mode, it only fetches the pages when the user requests it. The level of page monitoring can be specified as: monitor a single page, monitor a page and all the pages linked by it to the given depth, monitor the whole site. Monitoring an entire website is a difficult task for client machines, given that a large number of pages could be present in websites. Also, the granularity of page change cannot be specified and the application highlights every change that is detected.

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

- Flexible specification of versions: All the above systems compute changes between two successive pages. In WebVigiL the user can explicitly specify the pages that can participate in change detection.
- None of the above systems except Xyleme [27] use the ECA (Event-Condition-Action) paradigm for monitoring the web and notifying the changes. ECA Rules help in adding new functionality to the system seamlessly.
- Properties of monitoring requests can be inherited: The user has the option of specifying the monitoring request to be dependent on the status of other monitoring requests. One can specify the start/end of a request to be the start/end of another request.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

WebVigiL is a profile based change detection and notification system that monitors changes to web pages and notifies users based on their interests and preferences. A completely functional system had been developed whose modules were discussed in chapter 2. The goal of this thesis was to identify areas where the system could be improved to address performance, reliability, and scalability aspects of the system. The contributions of this thesis are:

- An version-caching strategy has been introduced to store the runtime objects of versions in a cache and reuse them when required. A replacement strategy called *utility policy* has been designed to manage the objects in version-cache. The effectiveness of this strategy is analyzed by comparing its behavior with traditional policies.
- A diff-patch approach to efficiently store versions of a page to reduce the storage growth of the repository has been analyzed and implemented to address the scalability of the system.
- Failure detection and recovery capability has been added to WebVigiL.
- A server-side fetch module has been introduced and its effectiveness in terms of fetching costs has been compared with the fetch module at WebVigiL mediator.

WebVigiL system now is a true server that can be started independently. Once started, WebVigiL server will be able to listen to requests from clients. The interface given to users for requesting sentinels is independent of the WebVigiL server. The server can

be gracefully shutdown when required for maintenance. A fail detection and a recovery system has been introduced into the system. One of the contributions of this thesis was to develop a plug-in module that can be controlled with configuration parameters.

8.2 Future work

In the current version-caching mechanism utility factor is estimated with respect to *a single sentinel*. This approach is chosen as obtaining the information of the exact requirement of a version in the system is difficult. If the fetch rules for sentinels are grouped, then a version required for a group of sentinels can be fetched only once. In such a scenario the utility of a version depends upon the sentinels in the group. So if the requirement of a version for a group is known the objects in the version-cache can be maintained by estimating the requirement for a group instead of a single sentinel. Server-side fetch mechanism is a way of moving the functionality of fetch module to a different system. WebVigiL can be extended to be a distributed system. Monitoring of sentinels can be performed by distributing the subsystems over different machines. The current change detection algorithms only detect appearance or disappearance of keywords, links, images etc. It can be improved to support changes with respect to numerical values like stock values (if the value goes down by x), flight deals (ticket price lower than $200.00) etc.

REFERENCES

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

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

[3] J. Jacob *et al.*, "Webvigil: An approach to just-in-time information propagation in large network-centric environments," in *Web Dynamics Book*. Hawaii: Springer-Verlang, 2003.

[4] N. Pandrangi *et al.*, "Webvigil: User-profile based change detection for html/xml documents," in *Proceedings 20th British National Conference on Data Bases*, Coventry, UK, 2003.

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

[6] J. J. Sharma Chakravarthy, Ajay Kumar Eppili and A. Sachde, "Expressive profile specification and its semantics for a web monitoring system," in *ER2004, 23rd International Conference on Conceptual Modeling Shanghai, China*, 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] U. diff utilities, "http://www.linuxforum.com/man/diff.1.php."

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

[22] Serialization, "Sunmicrosystems, object serialization specification sun microsystems inc. 2001."

[23] Crontab, "http://support.christianwebhost.com/kb/article.html?id=318."

[24] Push, "http://www.nysscpa.org/cpajournal/1997/1197/features/f341197.htm."

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

[26] Cygwin, "http://www.cygwin.com/."

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

[28] S. Z. Swarup Acharya, Michael Franklin, "Dissemination-based data delivery using broadcast disks," in *IEEE Personal Communications*, December, 1995, pp. 183–194.

[29] C. Mohan, "Aries: A transaction recovery method supporting finegranularity locking and partial rollbacks using write-ahead logging." in *ACM Transactions on Database Systems*, 1992, pp. 94–162.

[30] K. Rothermel and C. Mohan., "Aries/nt: A recovery method based on writeahead logging for nested transactions," in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989, pp. 337–346.

[31] G. Ramakrishna, "Database management systems, third edition," pp. 320–330, 2003.

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

[33] RSS, "http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html."

[34] R. Reader, "http://www.bradsoft.com/feeddemon/."

[35] Y. web alerts, "http://beta.alerts.yahoo.com/."

[36] G. web alerts, "http://www.google.com/alerts."

[37] W. application, "http://www.wysigot.com."

BIOGRAPHICAL STATEMENT

Ajay Eppili was born in Hyderabad, India, in 1978. He received his B.E.(Hons.) degree from Birla Institure of Technology and Sciences, India, in July 2000. In the Fall of 2001, he started his graduate studies in Computer Science at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington, in December 2004.