WebVigiL: Architecture and Functionality of a Web Monitoring System

Sharma Chakravarthy, Naveen Pandrangi, Jyoti Jacob, Anoop Sanka
{sharma,pandrang,jacob,asanka}@cse.uta.edu

January, 2003
WebVigiL: Architecture and Functionality of a Web Monitoring System

Sharma Chakravarthy, Naveen Pandrangi, Jyoti Jacob, Anoop Sanka
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76019
{sharma,pandrang,jacob,asanka}@cse.uta.edu

ABSTRACT
Data on the web is constantly increasing. Many a times, users are interested in specific changes to the data on the web. Currently, in order to detect changes of interest, users have to poll the pages and check for the changes he/she is interested in. Efficient and effective change detection and notification is critical in many environments where a lot of resources are wasted in monitoring changes to the web manually. In this paper, we present the architecture of WebVigiL [1], a change monitoring system, which efficiently monitors changes to the page on behalf of the user and notifies the changes in a timely manner.

WebVigiL is a general-purpose, server based information monitoring and notification system. WebVigiL handles the specification, management, and propagation of changes as requested by a user while meeting the quality of service requirements. We use the active capability in the form of event-condition-action (ECA) rules, and a combination of push/pull paradigm for change monitoring. In this paper, we present the semantics of our specification language, composite change detection, use of ECA rules for the run time management of sentinels, and a learning algorithm for fetching pages, that adapts to the actual page changes. We conclude with the implementation status of WebVigiL.

Keywords

1 INTRODUCTION
There are a number of situations where one needs to know when changes are made to one or more documents that are stored in a distributed (typically heterogeneous) environment. The number of documents that need to be monitored for changes is large and are spread over multiple information repositories. The emphasis here is on selective notification, i.e., changes are notified to appropriate persons/groups based upon interest (or profile/policy) that has been established earlier. Also, there should be a mechanism for establishing the interests/profiles/policies. Currently, change detection is done either manually or by using queries to check whether any document of interest has changed (since the last check). This entails wasted resources and at the same time does not meet the intended timeliness (where important) of change detection and associated notification. Also, quality of service is not considered at all.

Information retrieval in the context of the web is an example of the above problem. Change detection to structured documents (such as requirements analysis, detailed design, etc, in a large multi-year project) spread over heterogeneous repositories is another example. Different users may be interested in knowing changes to specific web pages (or even combinations there-of), and want to know when those changes take place. This will avoid periodic polling of the web to see whether the information has changed. Some examples are: students want to know when the web contents of the courses (they have registered for) change; users may want to know when news items are posted in a specific context (appearance of key words, phrases etc.) they are interested in. In general, the ability to specify changes to arbitrary documents and get notified according to user-preferred ways will be useful for reducing/avoiding the wasteful navigation of web in this information age. The proposed system – termed WebVigiL also provides a powerful way to disseminate information efficiently without sending unnecessary or irrelevant information. It also frees the user from having to constantly monitor for changes using the pull paradigm.

Today, information retrieval is mostly done using the pull paradigm, where the user is responsible for posing the appropriate query (or queries) to retrieve needed information. The burden of knowing changes to the contents of pages in interested web sites is on the user, rather than on the system. Although there are a number of applications (airlines, for example) that selectively send interested information periodically, the approach typically uses a mailing list to send the same information to all users. Other tools that provide real-time updates in the web context (e.g., stock updates) are customized for a specific purpose and have to be running continuously and underneath still uses a naïve pull paradigm to refresh the screen periodically.

In this paper, we demonstrate how techniques developed in the context of active databases can be exploited to provide solution to the broader problem of change detection in a distributed environment. WebVigiL is a system currently under development at UTA for providing an alternative paradigm for monitoring changes to the web (or any structured document). This paper discusses the architecture of the WebVigiL and the usage of Even-Condition-Action (or ECA) rules for fetching. We also present details of the change specification language, event-based fetch module that learns about changes and adapts its fetch policy, and a data-flow based change detection graph.

The architecture of WebVigiL is shown in Figure 1. User-defined sentinels are verified both syntactically and semantically prior to persisting their details in the

---

1 This work was supported, in part, 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).
knowledge base. Once a sentinel is validated, ECA rules are generated by the change detector module for the run time management of that sentinel. The fetch module fetches pages for all active (or enabled) sentinels, forwards them to the version management module for adding them to the page repository and notifies the change detection module. The change detection module detects the changes according to the specification and notifies the presentation module. All the modules have been designed for efficient execution and scalability requirements. The contributions of this paper are:

- Use of a new paradigm for monitoring information to structured documents (web is used in this paper)
- Expressiveness of change specification and its semantics
- Event-based fetching and a learning algorithm that adapts to actual page changes
- ECA rule generation and the dataflow-based approach to change detection.

The remainder of the paper is organized as follows. Section 2 introduces the notion of a sentinel, its specification, and semantics. In section 3 we describe the knowledge base used by all the modules. Section 4 discusses ECA rule generation and the framework for detecting various types of changes. In section 5 we discuss an event-based, adaptive page fetching algorithm and compare its effectiveness against both ideal and naïve algorithms. Section 6 covers related work and section 7 presents conclusions, implementation status of WebVigiL, and future work.

2 CHANGE SPECIFICATION

The Internet has evolved as an indispensable repository of information. The present day web user’s interest has extended from mere retrieval of information to monitoring the sequence of changes (to web pages) that are of interest. As the web pages are distributed over multiple large repositories, the emphasis is on selective and timely propagation of information changes. Changes need to be notified to the user in different ways and on different devices as specified in their profiles/policies. Hence, there is a need to define an expressive specification language to specify details, such as the web page(s) to be monitored, the type of change (keywords, phrases etc.), and the interval for comparing occurrence of changes. The user should also be able to specify how, when, and where to be notified, taking into consideration the quality of service (QoS) factors (timeliness, size vs. quality of notification) for example). In addition, the user may also specify a reference page with which a new page is compared for changes. WebVigiL provides an expressive language with well-defined semantics for specifying the monitoring requirements of a user, pertaining to the Web. Each monitoring request is termed a Sentinel. A specification language defined for this purpose should at the least support the following features:

- Specification of a sentinel in terms of previously defined sentinels. Also, start and stopping of a sentinel may be based on other sentinels. This provides a mechanism for tracking correlated changes.
- Duration/lifespan and the ability to disable and enable as needed. Again, the ability to use other sentinels allows for a lifespan specification that cannot be expressed by using temporal constants only.
- A suite of change types at appropriate levels of granularity that are of interest to a large class of users. For example, changes only at the level of a page may be overkill in many cases. One may be looking for changes in keywords (e.g., BCS rating) or phrases of interest.
- Ability to monitor a page based on the actual change frequency, or at a user-specified frequency. The specification of the actual change frequency relieves the user of knowing when the page changes and requests the system to do its best effort.
- Ability to specify detection of multiple types of changes on a page
- Notification frequency either as best effort or with predetermined frequency
- Multiple ways to compare changes (e.g., pairwise, every n, or moving n)

Taking these requirements into consideration, the semantics of the specification language for WebVigiL has been formalized. The syntax of the language is shown in Figure 2. Following are some scenarios and their representation using the sentinel specification language (SSL). We will use these as running examples in the rest of the paper.

Scenario1: Jill wants to be notified daily by e-mail for changes to links and images to the page “http://www.cnn.com” starting from December 2, 2002 to January 2, 2003. The sentinel (s1) for the above scenario is as follows:

Create Sentinel s1 Using http://www.cnn.com
Monitor all links AND all images
Fetch 2 day
From 12/02/02   To 01/02/03
Notify By email jill@aol.com
Every 4 day
Compare pairwise

Scenario2: Jill wants to be notified immediately for any change to the page “www.cnn.com/world” between the
start of s1 and the end of s1. The sentinel (s2) for the above scenario is:
Create Sentinel s2 Using http://www.cnn.com/world
Monitor any change
Fetch on change
From start (s1) To end (s1)
Notify By email jill@aol.com Every immediate
Compare pairwise

Scenario 3: Jill wants to monitor changes to images only once during the day (with reference to the previous day) on the page “www.cnn.com” and be notified as soon as possible. The sentinel (s3) for the above scenario is:
Create Sentinel s3 Using http://www.cnn.com
Monitor all images
Fetch 1 day
From 11/15/02 To 01/04/03
Notify By email jill@aol.com Every best effort
Compare moving

In the above, Jill wants to monitor changes at a fixed frequency irrespective of the actual change frequency.

The rest of this section elaborates on the semantics of a sentinel. For every sentinel, the system generates a unique identifier. The sentinel-target could be either a URL or a previously defined sentinel s_i. If the new sentinel s_n specifies the sentinel target as s_i, then s_n inherits its properties from s_i, unless the user overrides those properties in the current specification.

2.1 Sentinel Type

Meaningful change monitoring can be provided only if changes are detected at a finer level of granularity (e.g., phrases, keywords etc) on the content of the page rather than changes to the entire page. WebVigiL allows detection of customized changes and the sentinel type provides the semantics for the user to specify his desired type of change.

The semantics of sentinel type is given as:

\[ \text{Monitor} <\text{sentinel-type}> \]
\[ \text{sentinel-type}=\{<\text{unary op}>/<\text{change type}>\} \]
\[ \{<\text{binary op}>/<\text{change type}>\} \]

Change is computed between two versions of the same page with respect to the type of change specified. The sentinel-type is the change type t selected from the set T = {any change, all links, all images, all words except set of words, phrase: set of phrases, keywords: set of words, table: set of id, list: set of id, regular expression: \exp\} \). If V_1 and V_2 are two different versions of the same page, then Change C on V_2 with reference to V_1, is defined as:

\[ C_t(V_1, V_2) = True \text{ if the change type } t \text{ is detected as insert in } V_2 \text{ or delete in } V_1. \]
\[ False \text{ otherwise} \]

Primitive change is the detection of a single type of change C. For keyword change, the user must specify a set of words. For phrase change, a phrase is specified. For regular expression, a valid regular expression is given. A composite change comprises of a combination of distinct primitive change(s) specified on the same page, using one of the operators AND, OR, and NOT. The semantics of composite change formed by the use of an operator can be defined as follows (Note that \Lambda, V, and ~ are Boolean AND, OR, and NOT operators, respectively):

\[ \text{OR}: \text{Disjunction, denoted by } C_1 \text{ OR } C_2, \text{ of two primitive changes } C_1 \text{ and } C_2 \text{ specified on version } V_2 \text{ with reference to version } V_1 \text{ of the same page, is detected if either } C_1 \text{ is detected or } C_2 \text{ is detected. Formally,} \]
\[ (C_1 \text{ OR } C_2)(V_1, V_2) = C_1(V_1,V_2) V C_2(V_1,V_2) \]
\[ \text{where } t_1 <> t_2 \]

\[ \text{AND}: \text{Conjunction, denoted by } C_1 \text{ AND } C_2 \text{ of two primitive changes } C_1 \text{ and } C_2 \text{ specified on version } V_2 \text{ with reference to version } V_1 \text{ of the same page, is detected when both } C_1 \text{ and } C_2 \text{ are detected. Formally,} \]
\[ (C_1 \text{ AND } C_2)(V_1, V_2) = C_1(V_1, V_2) \Lambda C_2(V_1, V_2) \]
\[ \text{where } t_1 <> t_2 \]

\[ \text{NOT}: \text{NOT is a unary operator, which detects the non-occurrence of the given change type } t \text{ on version } V_2 \text{ with reference to version } V_1 \text{ of the same page.} \]
\[ (\text{NOT } C_t)(V_1, V_2) = \neg C_t(V_1, V_2) \]

For sentinel s1, the composite change specified is the conjunction of the primitive changes all links and all images. The change monitoring of all the created sentinels is done by the change detection graph (CDG), which uses the active capability. CDG and Change computation will be discussed in more detail in section 4.2 and 4.3.
2.2 Fetch

If the new version of a monitored page is fetched with a fixed frequency, then that version will be used irrespective of whether a change has occurred on the page or not. This kind of fetch is used when the user is interested in monitoring changes only with a fixed frequency. Otherwise, ideally, the new version should be fetched only when the page has been modified. The fetch attribute allows a user to specify when a new page should be fetched: \textit{Fetch \textless time interval\textgreater| on change}

- \textit{On change} option indicates that the system responsible for fetching the page as soon as it is modified.
- \textit{<time interval>} \textit{t_1} is a fixed fetch interval. \textit{t_2} can be in terms of minutes, hours, days or weeks and is a non-negative integer.

To ensure that the pages are fetched on modification, without possibly missing any changes, a heuristics-based fetch algorithm termed the Best Effort Algorithm (discussed in section 5.4) has been developed. This algorithm is used in the rule associated with the best-effort rule. On the other hand, for fetching pages at a fixed interval, an interval-based rule is used, which is discussed in section 5.3.

2.3 Sentinel Duration

The duration defines the lifespan of a sentinel and is the closed interval formed by the start time and end time of the sentinel, during which the change is monitored. This is defined as:

\textit{From <timepoint>| <from event>}
\textit{To <timepoint>| <to event>}

The \textit{From} modifier denotes the start of a sentinel \textit{s} and the \textit{To} modifier denotes the end of \textit{s}. The start and end times of a sentinel can be a specific time or can depend upon the attributes of other sentinels. Users can specify the duration as one of the following: (a) \textit{Now} (b) \textit{Absolute time} (c) Relative time (d) Event based time

- \textit{Now} keeps track of the current time, and hence the current time (time of sentinel specification) is used
- \textit{Absolute time} \textit{T} can be specified as a point on the time line
- \textit{Relative time} is defined as an offset from a time point (either absolute or event-based).
- \textit{We assume that events, such as the start and end of a sentinel can be mapped to specific time points and can be used to trigger the start or end of a new sentinel. Start of a sentinel can also be dependent on the active state of another sentinel and is specified by the event ‘during’}. During \textit{s_i} defines that a sentinel should be started in the closed interval of \textit{s_i} and the start should be mapped to \textit{Now}.

The semantic correctness of the specified or inherited start and end time of a sentinel has a bearing on the correctness of a sentinel’s execution. If the duration is not specified correctly, the sentinel will never start or continue forever without ending. For example, sentinel \textit{s_2} is dependent on sentinel \textit{s_1} to start its monitoring. But if \textit{s_1} has already started before the creation of \textit{s_2}, then \textit{s_2} will never start. Hence, the semantic check should capture such errors and notify the user. The sentinel is activated at the start time and is deactivated at the end time by the events and rules defined by the ECA rule generator module, using the concepts defined in [2, 3]. This is explained in detail in section 5.3.

2.4 Notification

Users need to be notified of detected changes. The mechanism selected for notification is important especially when multiple types of devices with varying capabilities are involved. The \textit{<contact options>} allows the users to select the appropriate mechanism for notification from a set of options \textit{O = \{email, fax, PDA\}}. The default mechanism is email. The notification has to be sent to the user taking into consideration the QoS constraints. The system should incorporate the flexibility to allow users to specify the desired frequency of notification. For example, in sentinel \textit{s_1}, Jill wants to be notified once in 4 days, irrespective of when the changes are detected. The semantics of notification frequency has been defined as:

\textit{Notify every best effort | immediate | interactive| <time interval>}

- \textit{<time interval>} \textit{is as defined in section 2.3}
- \textit{immediate} means to be notified immediately on change detection.
- \textit{best effort} is defined as notify ASAP (as soon as possible) after change detection. Hence, best effort is equivalent to immediate but will have lesser priority than immediate for notification.
- \textit{Interactive} is a pull based notification approach where the user pulls the detected changes as and when needed. A WebVigiL dashboard will be provided to the user to view and query the changes generated by his sentinels.

For sentinel \textit{s_2}, the user wants to be notified immediately when the change is detected and hence has specified a notification frequency of immediate. For sentinel \textit{s_3}, the notification frequency is specified as best effort. In the Change Detection Graph (CDG), each type of change on a particular URL is a node (section 4.2). When a change is detected on the particular change type node in the CDG, all the users interested in this change type on the specific URL are notified.

2.5 Compare Options

Changes are detected between two versions of the same page. Each fetch of the same page is given a version number. The first version of the page will be the first page fetched after a sentinel starts. Given a sequence of versions \textit{V_1, V_2 ……V_n} of the same page, the user may be interested in knowing changes with respect to different references. By default, the page previous (based on user-defined fetch interval where appropriate) to the current page is used for change detection. For example, a user may want to monitor changes between every \textit{n} versions of the page. In order to facilitate this, the compare option can be selected from a set \textit{P = \{pairwise, moving n, every n\}} and is specified by: \textit{Compare <compare options>
The details of a sentinel need to be stored (in a persistent and recoverable manner) as several modules use this information at run time. For example, the change detection module detects changes based on sentinel information such as the URL to be monitored, the change and compare specifications, and the start and end of a sentinel. The fetch module fetches the pages based on the user specified fetch policy. The notification module requires appropriate contact information and notification mechanism to notify the changes. User information, such as the sentinel installation date, and the page versions for change detection and storage path of detected changes also need to be stored to allow a user to keep track of his/her sentinels.

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

4 ECA RULE GENERATION

Every valid user request arriving at WebVigiL, initiates a series of operations that occur at different points in time. Some of these operations are: creation of a sentinel (based on start time), monitoring the requested page, detecting changes of interest, notifying the user(s) of the change, and deactivation of sentinel. In WebVigiL, for every sentinel, the ECA rule generation module generates ECA rules [4, 5] to perform some of these 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 paradigm has been used for monitoring the database state in active databases and as a stand-alone concept for monitoring objects in applications (both centralized and distributed [6]). As part of the Active Object-oriented system [7, 8], a local event detector (LED) has been developed as a library that can be used to declare events and associate rules to be executed when events occur. Primitive events (as method executions) and temporal events (both absolute and relative time), as well as composite events (And, or, seq, and periodic used in WebVigiL) are supported in LED. ECA rules provide an elegant mechanism for supporting asynchronous executions based on events (temporal or otherwise).

In this section, we will discuss how ECA rules along with the LED (local event detector) are used for: i) generating fetch rules for retrieving pages, ii) detecting events of interest and propagating pages to detect primitive and composite changes, and iii) for activation and deactivation of sentinels.

4.1 Activation/Deactivation

During its lifespan, a sentinel is active and participates in change detection. A sentinel can be disabled (does not detect changes during that period) or enabled (detects changes). By default, a sentinel is enabled during its lifespan. The user can also explicitly change the states of the sentinel during its lifespan. The start/end of a sentinel can be time points or events (as explained in section 2.3).

When a sentinel's start time is now, it is enabled immediately. But in cases where the start is at a later time point or depends on another event that has not occurred, we need to enable the sentinel only when the start time is reached or the event of interest has occurred. In WebVigiL, it is the responsibility of ECA rule generation module to create appropriate events and rules to enable/disable sentinels. We achieve this as follows. Consider the scenario where s1 is defined in the interval [12/02/02, 01/02/03]. At time 12/02/02 sentinel s1 has to be enabled. Following are the events and rules that are generated to enable sentinel s1 at compile time:

Event Temp1 = createTemporalEvent(12/02/02)
Event start_s1 = createEvent("start_s1")
Rule T1 = createRule(Temp1)
Event Fetch_s1 = createPeriodicEvent (Start_s1, 2, End_s1)
Fetch_s1 is a periodic event created with "start_s1" as the start event, the frequency of page fetch, and End_s1 as the end event. The rule associated with it handles the fetching
of pages for s1. A rule associated with an event is fired when the event is triggered. More than one rule can be associated with an event. When event Temp1 is triggered at the specified time point, rule T1 is executed, which in turn raises the event start_s1. Triggering of the event start_s1 activates the sentinel s1 by activating the periodic event used for fetching the pages of URL specified in s1. Now, if another sentinel s2 which is defined over the interval [start(s1), end(s1)] arrives, the following events and rules are generated in order to enable s2:

Event start_s2 = createEvent("start_s2")
Rule r_start_s2 = createRule(start_s1)

Here we are associating the rule r start s2 to the event start_s1, which was created at the arrival of sentinel s1. This rule actually raises the start_s2 event to activate the periodic event associated with s2. In this manner, ECA rules are used to asynchronously activate and deactivate sentinels at run time. Once the appropriate events and rules are created, the local event detector handles the execution at run time. By enabling/disabling of sentinel we mean addition/deletion of that sentinel to the change detection graph that is detailed in the next section.

4.2 Change Detection Graph (CDG)

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

The change detection graph for the sentinels s1 and s3 (defined in section 2) is shown in Figure 4. The different types of nodes in the graph are as follows:

- **URL node**: A URL node is a leaf node that denotes the page of interest. The number of URL nodes in the graph is equal to the number of distinct pages the system is monitoring at any particular instance of time.

- **Change type node**: All level-1 nodes in the graph belong to this category. This node represents the type of change on a page (all words, links, images, keywords, phrases, table, list, regular expression, any change).

- **Composite Node**: A Composite node represents a combination of change types. The possible combinations are explained in section 2.1. All higher-level nodes (> level-1) in the graph belong to this type. Currently we support composite changes on a single page. We plan on extending this to multiple pages.

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

Consider the scenario where a page p1 changes twice a day and sentinel s1 is interested in both the changes taking place each day whereas sentinel s2 is interested in changes between two days. In this case both the sentinels are interested on the same type of change on the same page but with different requirements (versions of page). Hence additional information has to be maintained at the change type node in order to differentiate the change detection and change propagation among the sentinels. Consider the change detection graph shown in Figure 4. The arrows in the graph represent the parent-child relationship among the nodes. The subscription information is maintained in the list associated with each node. Sentinel s1 specifies a composite change consisting of links AND images. Sentinel s1 should be notified only when there is a change to links and images. Since change is computed only at level-1 nodes, s1 should have representation of itself at the constituent change type node. In order to facilitate this, a proxy sentinel s_end with the same properties as that of s1 is created on the links and images node. The composite node AND notifies the creator of s1 only when it receives notifications from both of its constituent sentinels. This is illustrated in Figure 4. At run time whenever a request for monitoring a page arrives, the corresponding nodes are created (if they do not already exist) and the sentinel is added to the change detection graph.

![Figure 4: Change Detection Graph](Image)
4.3 Change Computation
A detection algorithm associated with each change type node computes changes between two versions of a page with respect to that change type. For a change to be detected, the object of interest is extracted from the given versions of the page depending upon the change type. Change detection algorithms have been developed to detect different types of changes to HTML and XML pages. The change types currently supported are: links, images, all words, keywords, phrase and regular expressions. Change to links, images, words and keyword(s) is captured in terms of insertion or deletion. Object identification, extraction and change detection is complicated for phrases. For identifying an object (phrase) in a given page we use the words surrounding it as its signature. We assume that these words are relatively stable. WebCQ [9] uses the concept of a bounding box to tackle this problem. For phrases, an insert or delete indicates appearance or disappearance of the complete phrase in the page.

In section 4.1) and

4.4 Fetching
For monitoring a page, after every time interval t, we check for any change in page properties, such as the last modified time stamp of the page and then actually fetch the page if there is a change. Hence we need a mechanism for triggering the monitoring requests at a given time point after the lapse of interval t. We solve this problem by using the notion of a periodic event [2]. A periodic event is an event that repeats itself with a constant and finite amount of time. The specification for a periodic event is PeriodicEvent (E1, [t], E3) where E1 and E3 are the events (or time specifications) that act as an initiator and a terminator, and t is the time interval. In WebVigiL, we use periodic events for triggering the monitoring request. Here E1 and E3 are the start and end events of a sentinel (defined in section 4.1) and t is the interval at which the page should be monitored. To actually fetch the page, we associate a fetch rule with this periodic event. Here we start with a time interval, start event and end event we generate a unique fetch rule with it. Hence whenever a periodic event occurs, the fetch rule associated with it fetches the page. For sentinels that explicitly specify the polling interval, we generate a periodic event and associate a fetch rule with it. The interesting and difficult case is when the user expects the system to notify him/her as and when the page changes. In such cases, the system is required to tune its polling interval with the change frequency of the page. Here we generate one periodic event with a fetch rule. This rule achieves the required tuning by changing the interval of periodic event. The functionality of this rule is discussed in detail in the following section.

5 EVENT-BASED FETCHING
WebVigiL monitors only those pages that are registered with it. For this purpose it has to fetch the pages when a change in metadata (such as the last modified time stamp or checksum) is detected. In this section we will discuss the criteria for fetching, the paradigm being used for fetching, and present an adaptive algorithm to tune the polling interval to change interval and experimentally analyse its effectiveness against the ideal and naïve approaches.

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

Naïve Method: Since the pattern of page change (intervals at which the page is changing) is not known, one approach could be to fetch the page at a pre-defined interval. This approach will not miss any changes, provided the change interval is greater than the fetch interval. But for those pages that are changing infrequently (i.e., at intervals greater than the fetch interval), this approach will result in excessive network connections and data transfers, as the nature of change is not taken into consideration in determining the fetch interval.

Adaptive Method: In this approach, we start with a time interval for fetching, and use the history of change patterns on the page, as observed and recorded by the previous fetch cycles to determine and tune the time at which the page should be subsequently fetched. Initially we start with a predefined frequency as in the Naïve method and try to converge on the actual change interval. This approach does not catch all the changes but detects significant number of changes with reduced number of network connections and concomitant data transfers.

5.2 ECA Paradigm for Fetching
Event-Based Fetch module is responsible of monitoring and fetching pages that are of interest to the sentinels registered with WebVigiL. Periodic events are defined and rules are associated with them as discussed earlier. Whenever a periodic event occurs, the corresponding rule
is fired, which then checks (condition part of the rule) for change in metadata of the page and fetches the page (action part of the rule) if there is a change in metadata. Thus the periodic event controls both the polling interval and the lifespan of the fetch process. The meta-data of the page is taken into consideration for fetching. By meta-data of the page, we mean page properties such as the page size, last modified time stamp and checksum of the page. A fetch cycle for a page is triggered only when there is a change between the meta-data of the current version of the page to that of the previous version. Depending upon the nature (static/dynamic) of page being monitored the complete set or subset of the meta-data is used to evaluate the change.

For static pages, HTTP HEAD request is used to obtain the meta-data of the page. Change in time stamp of the page with an increase or decrease in page size, is flagged as change, and the page is fetched. In cases where time stamp is modified, but the page size is unchanged, HTTP GET request is used to retrieve the page and the checksum of the page is calculated. The page is added to the page repository only if the calculated checksum differs from the checksum of the previously cached copy of that page. For pages that are not provided with last modified timestamp such as dynamically generated pages or cases where previous attempts to retrieve page properties have failed, HTTP GET request is used to retrieve the page. Change is then flagged by calculating the checksum.

5.3 Types of Fetch Rules
A fetch rule is created and used for polling the page of interest specified in the sentinel. As per the change specification, a user can specify a sentinel for fetching with two options (a) On Change (b) or Interval Based. Based on the option specified in the sentinel, Event-Based Fetch module generates a Best-Effort (BE) Rule or an Interval-Based (IB) Rule. These Rules differ in the way they handle the “t” (“fetch interval”) of the periodic event.

Best Effort Rule: In situations where the user has no information about the change frequency of a page, it is necessary to tune the fetch frequency to the actual change frequency of a page. BE Rule uses a best-effort Algorithm to achieve this tuning. In the best-effort algorithm (BEA), the next fetch interval \( P_{next} \) 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. Clearly, the effectiveness of the algorithm depends on the accurate estimation of the fetch interval. Event-Based Fetch Module generates a BE Rule BE for every unique page \( u_i \), and maps other sentinels with fetch option “on change” on \( u_i \) to the generated rule BEi. For sentinel \( s2 \), Jiff specifies the fetch options as “On change” for the URL “http://www.cnn.com/world”. The fetch module checks if there was a BE rule created for this particular URL; if not present, it creates a BE Rule on this URL with the start event on the start time of the sentinel.

Interval-Based Rule: The user can explicitly specify a fetch frequency. For example, Don may know that a page is changing every 4 hrs starting at 9.00 am and hence can specify a sentinel to start monitoring the page with a fetch interval of 4 hrs. For this, a periodic event whose periodicity (interval t) equal to the given interval is created and an IB rule IBi is associated with it to fetch the page. As a result there will be more than one IB 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).

A BE rule and many IB rules can be set for the same page. Thus, there can be situations where both a BE rule and an IB rule fetch the same version of the page resulting in multiple copies of the same version at page repository. To avoid this situation, we synchronize the fetching with respect to the last fetch version of the page. A rule initiates the fetch process only when, there is no version \( V_i \) of the page \( u_i \) with Last-Modified-Timestamp (LMT) equal to LMT of the page it is required to fetch.

5.4 Best-Effort Algorithm (BEA)
Let \( P_{next} \) denote the minimum period after which the BE rule should fetch the page if it has changed in the interim. A BE rule computes the \( P_{next} \) value based on the observed rate of change to the page. Rapidly changing pages result in a smaller \( P_{next} \), whereas infrequent changes require less frequent fetches of the meta data, and hence, a larger \( P_{next} \). Clearly, the success of the BE rule depends on the accurate estimation of the \( P_{next} \) value. To tune \( P_{next} \) to the change interval of the page, we take into account (a) the most recent observed intervals with which the page is changing and (b) a static lower bound \( P_{min} \) so that \( P_{next} \) values are not set too low. Changes observed to a page can be categorized into (a) no change (b) constantly changing or (c) randomly changing. Prediction or calculation of the \( P_{next} \) value depends on the nature of change observed during fetching.

In what follows, we use \( P_{current} \) to denote the interval with which the BE rule is fetching the page currently and \( H_{next} \), \( H_1 \ldots \ H_t \) to denote the change intervals observed during fetching. \( P_{next} \) is adaptively computed as:

- No change: \( 2*P_{current} \), if there were no change to the page properties for t cycles, where \( 0 < \alpha < 4 \).
- Constant change: \( H_1 \), A page is said to be changing at a constant interval when \( H_1 = H_{next} \). To handle situations where the actual change interval is stepped up or stepped down from that observed, we take a pessimistic approach. \( P_{next} \) is recalculated after every \( \beta \) cycles assuming the change to be random, \( 0 < \beta < 4 \).

For random changes, a value \( P_{estimate} \) with a weight factor \( w \) is used. \( P_{estimate} \) is a value derived from the history of change intervals observed i.e., \( P_{estimate} \) is a value computed from history, typically \( P_{estimate} = \sum H_i / n \) where \( i = 0 \ldots n \) and weight \( w \) initially (0.6) is a measure of the relative preference given to recent and old changes, and is adjusted by the system so that more recent changes affect the new \( P_{next} \) more than the older changes. Determination of \( P_{next} \) for pages that are varying with a random interval requires some computation. Here estimation of \( P_{next} \) not only depends on \( P_{estimate} \) but also on \( P_{current} \) and \( P_{min} \).

- Max \( ((P_{next} > P_{current} > P_{estimate} > w : P_{estimate}), P_{min}) \), \( P_{next} \) value for random changes depends on the amount of history considered for the calculation of \( P_{estimate} \). A page that changed sparsely in the past and is changing at smaller intervals now, will result in having a \( P_{estimate} > P_{current} \). In such cases, \( w \) is used to give more
There are many factors that influence the adaptability of BEA. In cases where a page changed more frequently in past and is now changing sparsely, the history will be large. Thus the amount of history considered for determination of $P_{next}$ plays an important role. Lower values of $\beta$ will result in excessive computational overhead even though the page was changing with the observed interval. Smaller values of $\alpha$ will result in large $P_{next}$, thereby increasing the chances of missing changes in the interim. Currently $\alpha$ and $\beta$ are set to 3 and 2 cycles. Initially the interval of a periodic event is set to $P_{min}$ (currently equal to 2), when the periodic event occurs, the BE rule is fired which uses BEA to change the interval of the periodic event to adapt to the change interval.

### 5.5 Experimental evaluation of BEA

In this section we evaluate the effectiveness of the proposed BEA against the other approaches with respect to total fetch cycles, number of times the fetch interval was tuned to the actual change interval, and the number of changes captured. A testbed on a remote webserver is used for the experiments. Test cases were generated to change a page with various change intervals. For evaluation, we categorize the test cases into (a) Fast Random (RFast#), page changing with small intervals (between 2-10 seconds) (b) Constant Change (Cons#), changing with a fixed interval (5 seconds) and (c) Random change (Rand#), changes randomly i.e., it may change with small intervals or constant interval or with large intervals (between 10-30 seconds). "#" Indicates the number of times a page changed. Hence the case Random240 indicates that the page changed (with a random interval) 240 times.

#### 5.5.1 Fetch Cycles vs. Changes Captured

Table 1 shows total number of fetch cycles taken by each approach for every test case and the number of changes captured for that many cycles. Consider the Rand240, ideally 240 changes should be detected in 240 fetch cycles. Naïve approach took 800 cycles to detect 240 changes while BEA took 343 cycles to detect 224 changes (while missing the rest). The same trend is true for all cases. So BEA captures more than 90% of changes using 40% fetch cycles as compared to the naïve algorithm. Its performance is not far from the ideal case as well.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Fast Random 240</th>
<th>Cons240</th>
<th>Rand240</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEA</td>
<td>240</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>Naïve</td>
<td>362</td>
<td>216</td>
<td>299</td>
</tr>
<tr>
<td>Naïve</td>
<td>583</td>
<td>240</td>
<td>600</td>
</tr>
</tbody>
</table>

#### 5.5.2 Tuning of fetch interval

Let $r$ be the interval between two consecutive changes, we define convergence $c$ as $1/n$, where $n$ is the number of times we polled in that interval $t$. Ideally, $c$ should be equal to one. Hence the value of $c$ lies between 0 and 1. The effectiveness of an approach in tuning the fetching interval to the change frequency can be measured in terms of number of number of intervals in which convergence $c$ was 1. Figure 6 shows the total occurrences of each convergence $c$ of BEA and Naïve for two test cases RFast240 and Rand240. Consider the Rand240 case, where BEA polled 110 cycles with $c = 1$. Ideally it should have fetched 240 times. Hence 45% of the time BEA tuned its fetching frequency to the change frequency, where as Naïve fetched only 13% of the time with the change frequency. (Naïve fetches with a minimum frequency, without any tuning. The 13% times it polled with change frequency was mere coincidence).

The cases presented here and others performed (not presented due to space constraints) indicate that the BEA algorithm is doing far better than the naïve algorithm for different rates of page changes. In fact, the BEA is converging to the ideal case. We are currently trying to improve BEA further by computing $\alpha$ and $\beta$ dynamically.

### 6 RELATED WORK

Considerable research has been done in active database [10-12]. In WebVigiL, the active capability is used for the run time management of sentinels, as without an event-based approach, asynchronous monitoring is not possible. Also, it would have been very difficult to incorporate the learning algorithm into a framework that is not event based. Several tools have been developed, and are currently available for some of the problems addressed in WebVigiL. AIDE developed by AT&T [13] displays the difference between two html pages. Changedetection.com [14] allows users to register their request and notifies them when there is a change. The change is always detected at the page level and not at the granularity chosen by the user, as in WebVigiL. Mind-it [15] and WebCQ [9] both support customized change detection and notification. Both of these systems track changes to a finer level of granularity in a page. Mind-it does not support change specification on combinations of change types within a page (e.g., phrase change and a link change) as permitted in WebVigiL. In WebCQ, changes are detected only between subsequent versions of the page unlike WebVigiL, which provides flexibility in specifying the reference page. In Xyleme [16, 17], the idea of active paradigm is being used for detecting
changes by evaluation of continuous/monitoring queries on XML/HTML documents. The focus is on the subscription language and continuous queries. Lifespan and dependent sentinels are not supported. Furthermore, Xyleme detects changes based on the structure of the document. Change detection algorithms are simpler as WebVigiL detects changes on the information seen by the user (independent of the structure). Finally, unlike Xyleme, WebVigiL supports both user-specific frequency and on change (for page fetches). WebVigiL offers user-centric specifications in terms of on change (for fetching) and best effort (for notification) that are important from users’ viewpoint. The adaptive push/pull [18] approach evaluates the effect of various approaches (push, pull, and combinations there-of) from the point of view of propagating changes from server to the client. WebVigiL addresses a different problem – that of change detection based on an expressive user specification. Finally, the use of ECA rules and the optimisations that accompany its usage are novel to WebVigiL.

7 CONCLUSION

WebVigiL is envisioned as a complete system that allows monitoring and notification of changes to structured documents in a distributed environment. It is being developed as a modular system where components can be replaced relatively easily. For example, to move from web to structured documents (such as SGML), only the change detection algorithms need to be replaced. The other goal is scalability and efficiency of individual modules.

In this paper we have presented the architecture of WebVigiL along with the details of the change specification language, generation of ECA rules, event-based fetch module and an adaptive algorithm. Sentinel validation has been completed along with change detection for various types of changes. The BEA has been implemented and tested. The design of CDG is complete. A prototype will be available in the next couple of months that will support the system presented in this paper. We are currently investigating the presentation module and the versioning modules.

8 REFERENCES


