Temporal HTML

By

Lina Liu

M.Sc. Harbin Engineering University, 1995

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming to the required standard

Supervisor: Dr W.W. Wadge

ABSTRACT

Temporal HTML (THTML) as presented in this thesis is an extension of HTML -- a high level authoring language for World Wide Web documents. THTML incorporates temporal logic into HTML to provide an efficient solution for authoring and maintaining *time-sensitive* web sites. In THTML, the same URL request may result in different HTML pages for different request times. This request time may be the local time that the reader sends the request, or it may be sometime in the past or even the future. The HTML page sent in response to the request is an instance of the page determined by the particular time context, i.e. the reader's request time. The instance is generated using the THTML source files for each web component whose time interval stamps most closely approximate the time specified. With *server push* technology incorporated into the design, the THTML server periodically re-instantiates the previously requested URLs in each web session and re-issues the updated instances as time elapses. THTML 1.0 implemented the above design partially.

CONTENTS

LIST of FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr.W.W.Wadge, for his support, both financial and intellectual. THTML also originated from him. I also thank ISLIP 17 conference, on which my imagination is inspired about the high level design of THTML. I would also like to appreciate my dear sister, Mom and Dad, for their support and courage for decades.

I would like to appreciate Gordon Brown for his free IHTML 2.0, on which I build the THTML 10. Finally I would like to thank Som Tang, Micheal Ko and Paul Swoboda for their help in proofreading the thesis.

Chapter 1

Introduction

1.1 Introduction to Temporal HTML

Since Web technology provides an integrated presentation of all types of information (text, picture, video and audio) on a single screen, individuals and institutions are encouraged to establish their own Internet presence. This multi-media environment can consist of text, graphics, audio, video, three-dimensional models or any other information that a computer can process.

Web technology is based on 2 protocols, namely HTML and HTTP. Based on the TCP/IP Internet layer, HTTP dominates the protocols defined by W3C for the World Wide Web. In HTTP, a set of negotiations is defined between the client and server so that actions performed at the client can be interpreted by the server and the correct response can be sent to the client. For example, the *get* method is defined to retrieve certain resources at the server site. HTML (hypertext markup language) is normally considered a high-level authoring tool to write web container documents that hold different types of multimedia information. Web browsers can interpret the different types of data and display them on the screen. HTML is actually a protocol implemented on top of HTTP for presenting information on the web. It features an *archor* system by which any web page can refer to other pages, without any knowledge of the contents of the other page. It gives the web a better look and feel than that of a static electronic library.

Although HTML serves many applications very well as a high-level authoring language for the World Wide Web, it provides no temporal support for a site. With the rapid development of web technology nowadays and the appearance of more time sensitive applications on the web, it is no longer sufficient to provide only static information. Creating a *dynamic* site becomes critical for success. The non-temporal feature of conventional HTML is a serious hindrance to creating a more dynamic World

Wide Web. Some of the problems are concluded as follows:

1. In addition to retrieving the current version, web clients sometimes wish to retrieve historical information. This occurs on sites related to the news or stock market where retrieval of information based on time is necessary. However, the existing HTML protocol only supports keeping one version of a document in the system.
2. In time critical web applications, such as the stock market, information being viewed at one moment might become stale in the next. HTML does not provide a mechanism for synchronizing the server and clients, so clients will not be informed of changes to the information.
3. The demand for web authors to create *dynamic* rather than *static* sites is increasing dramatically. They need to decide *when* items should be posted, and not just *what* should be posted. The existing web servers do not have a mechanism selecting the file corresponding to a given time. The author has to manually update the site to make it appear dynamic from the client's perspective.

With the above demands for temporal support on the web, a lot of effort is being devoted to this area at research institutions and companies such as Netscape and Microsoft. As a result, many HTML enhancements have appeared such as *JavaScript*, *DHTML*, *XML* etc.

Although they improve the dynamic features of the web and make the web more interactive, none of them addressed the temporal demands of the web in a natural way. For example, *XML* provides developers with their own special purpose tags that can be defined for a variety of purposes. *DHTML* extends HTML by allowing multimedia applications to run on a user's desktop without interaction with the server. It also supports interactive databases and documents. However, the above enhancements don't provide a natural and efficient mechanism to allow servers to support complicated *temporal* phenomena in web applications.

With the introduction of Temporal Logic (TL), THTML provides an efficient solution for most temporal demands of a web site. It has great potential for adoption in

complicated time-critical web applications such as the stock market, newspaper, web commercials etc.

THTML enhances the following features to HTML:

1. THTML provides a mechanism to allow one to define arbitrary versions for the same HTML file. It also allows one to define versions for parts of a page.
2. THTML also allows the author to define multiple instances of an HTML page, each of which has a time interval stamp. This time interval stamp determines what content should be posted based on the URL request and the request time from a client.
3. A time point can be specified by the client to indicate the specific instance of an HTML page it requires. The time can be a *relative* point to now such as yesterday or the day before yesterday, or an *absolute* time point such as May 3rd, 1998 12:12:12, etc.

1.1.1 THTML Solution

THTML is a web-authoring tool intended for time sensitive web sites.
Time critical web sites have two basic requirements. One is that the page being viewed by the reader (the people who access or browse the site) is dependent on the time. For example, on a course web site, an instructor often wants assignment one posted during the first week. The solution of the assignment is posted in the second week along with the problems for assignment 2 and so on. The course notes should also be posted at appropriate times to match the schedule of teaching. In a newspaper site, different content should appear every day, and sometimes the reader would like to view yesterday's page as well as today's page.

The other requirement is that the content can be updated at customizable intervals, preferably automatically. This is often critical for stock markets where a user would like to see the most recent content of the site.

THTML fulfills the two requirements with a solution based on discrete temporal logic. THTML allows a single source file to specify multiple *instances* of the

corresponding page – different versions whose layout and content vary with time. Every request for a particular page in a THTML site is accompanied by a *request time*, which specifies the instance of the page that the reader wishes to view. The request time is often the current time but may also be a time in the past or even the future.

The *author* can specify multiple instances for a *source file*, each instance associated with an *interval label*, in which the author can specify a time period or time point or sets of time periods or time points. When the server receives a request for a URL from a

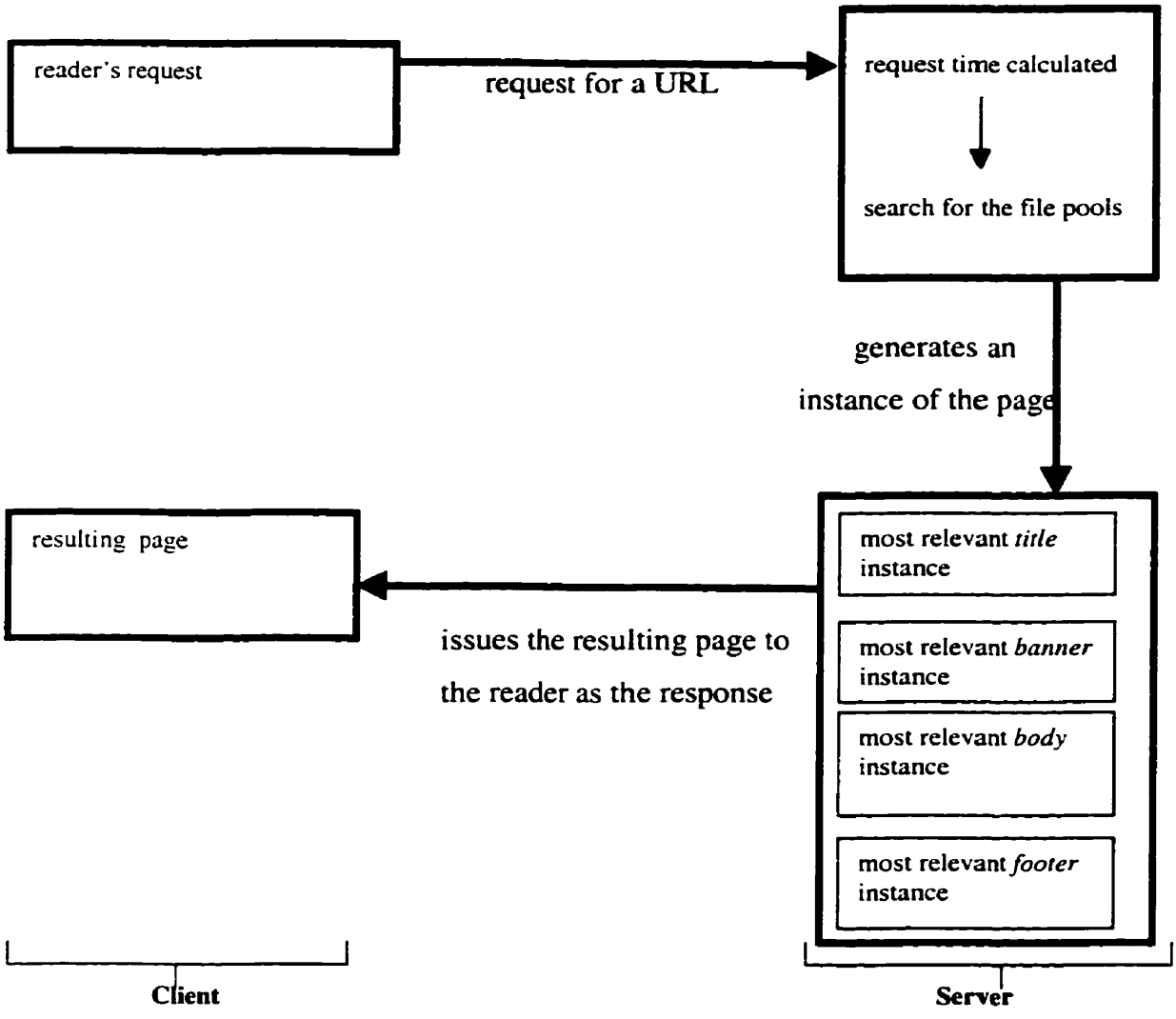


Figure 1.1 Yesterday's Instance of a URL

reader, the *instantiation* (the process of taking the general definition of a time-varying

page, combining it with a particular time-point and producing the instance) of the source file begins. The page of the URL is configured dynamically with the THTML source file of each component whose time interval label is most relevant to the request time according to *the temporal versioning algorithms*. The page will be issued to the client when it is created, and discarded afterwards. The *dynamic configuration* and the *temporal versioning algorithm* guarantee that the resulting page sent to the client is an instance at the time context specified in the reader's request time. Suppose, that a reader accesses a newspaper URL which has the following components: the title, banner, body and footer and requests yesterday's instance. The instance of each component corresponding to the reader's request time, namely, *the local time the server receives the request minus one day* in this case, constitutes a page for the requested URL and it will be issued to the reader as the response. An instance of each component is generated using the THTML source file that is most relevant to the request time.

Further, web pages change as time elapses, the instance a user is viewing may become obsolete. With push technology, synchronization can be achieved between the client and the server. After the initial request for a URL, the server sends updated instances to the *reader* at frequencies that the *reader* specifies.

1.2 The Relationship between IHTML and THTML

The implementation of THTML is based on Intensional HTML 2.0, which supports sites having multiple user-defined versions. THTML 1.0 extends it by allowing users to create temporal variants of each arbitrary user-defined version and associate time with each temporal variant.

Because THTML adds a time dimension to the IHTML version space and specifies more time related version control algorithms to eliminate temporal version ambiguity, THTML makes the web site more dynamic and time-sensitive.

1.3 Terminology

A few terms are used in particular ways in this thesis and the meanings are

explicitly declared here:

Readers-- the people who access/browse the web.

Authors-- the people who write the source for the web site.

Source files-- the source which the author writes and which generates pages and components.

Time point -- a complete temporal context, a particular point in time.

Instance—the version of a page/component/site corresponding to a particular time point.

Instantiation – the process of taking the general definition of a time-varying page, combining it with a particular time-point and producing the instance.

Time interval stamp—the time periods or time points specified at the beginning of a THTML source file, which is the basis for determining the most relevant instance to the reader's requested time. It is distinguished with a tag pair **<cron>** and **</cron>**.

Temporal versioning algorithm—the rules for determining the most relevant instance of a source file.

1.4 The Structure of the Thesis

The relevant background knowledge is introduced in Chapter 2, including the Intensional Logic based version control approach. Knowledge needed to implement THTML is also introduced, namely the HTTP protocol and the client/server model for communicating over the Internet. Chapter 3 examines THTML from three different perspectives: Language Design, System Design and Algorithm Design. Further enhancements to each design aspect are also discussed. Chapter 4 shows how to implement THTML on a web server using the *Apache 1.2.5 API*. Chapter 5 concludes the thesis and proposes further directions for research.

Chapter 2

Background

2.1 Intensional Logic

In natural language, there are many situations where the semantics of a sentence depends on implicit contexts. For example, in the phrase

Today's temperature is five degrees less than yesterday's temperature,

the value of today's temperature relies on the value of yesterday's temperature at the same place, therefore, the sentence "*today's temperature is 5 degrees*" may or may not be true.

Although implicit context sensitive expressions have for a long time been considered a non-mathematical and illogical aspect of natural language, logicians have tried to capture them in formal methods. Intensional Logic is the result. It is the study of context sensitive properties using formal methods. Intensional Logic uses predicate calculus to represent these context sensitive expressions.

An Intension is an entity that depends on implicit contexts. For example, temperature is an intension that is variable in time and space. The time and place that is implicit in a sentence determines the temperature's value. In IL, *intension* and *extension* are distinguished for an expression. It is based on the distinction between *sense* and *denotation* of it. *Intension is the entire concept of an expression—what we, at some level, intend when we write it, while the extension denotes the particular object it in fact denotes (or currently denotes). For example, the expression " the President of France" denotes currently Jacques Chirac; but no one would claim that M Chirac somehow sums up the whole concept of the French presidency*[18]. As we can see, an intensional object may have different extensions at different implicit contexts. *For example, the French constitution specifies that the French President is directly elected; this is not the same as specifying that M Chirac be directly elected*[18].

The basic concept in IL is the *possible world semantics* proposed by Montague.

Simply speaking, it refers to a space of indices that an entity varies over -- the implicit contexts. A simple example is the World Wide Web (WWW). The web can be perceived as an indexed family of pages, the indices being the URLs. (URL's are required by the protocol to be unique, a necessary condition for indexing.)

Another example is the *chessboard* world, which is composed of all possible configurations of pieces on the board. As one can see, there may be many possible worlds in a given scenario. Using particular rules, one can switch from one possible world to another. (Modal logic is a variant of IL, in which people study the necessity and possibility of a statement in some or all the possible worlds, and the accessibility among them. Refer to [1] for more information in Modal Logic.)

2.2 Temporal Logic-An Instance of Intensional Logic

Temporal Logic (TL) is an instance of IL in which the collection of contexts models a collection of time points. Many areas in computer science adopt it extensively because high-level languages need a way of describing behaviors of complex systems in time. TL is useful for reasoning about a changing world. Since the temporal order of actions/events can be described, they are useful for representing dynamic systems. TLs are widely used in the specification and verification of reactive systems and in applications where the concept of time is central, such as temporal planning, temporal representation and temporal databases. Consequently, programming languages that provide access to such temporal concepts built on temporal logic such as *Chronolog* have a wide range of applications.

Further, the logic power of the language allows people to express complex temporal properties of the system. For example, if Discrete Temporal Logics (DTL's) are used, where time is represented as a sequence of distinct moments, a temporal formula can be used to effectively represent the individual steps of an execution. THTML adopts DTL so that the time context of a page can be efficiently reported at discrete moments.

2.3 An Application of Intensional Logic to Version Control

An Intensional Programming Language (IPL), which can be textual or visual,

retains two aspects of IL. First, at the syntactic level, there are context-switching operators, called intensional operators, that can switch intensional contexts from one possible world into another. Second, at the semantic level, possible world semantics must be provided for each intensional operator. THTML is a web authoring tool that interprets version phenomena on the web using IL and provides an efficient method to generate instances of a HTML page and allow clients to switch among different possible worlds. THTML uses the *eduction* model, which is a tagged, demand-driven dataflow model. In this model, an intension is computed lazily. Each demand is tagged by an index, and is evaluated by the extension at that index, the demand and their results flow as packets in an asynchronous network. In the eduction model in THTML, the server will generate specific instances and respond to clients only if the client requests it at a given time.

2.3.1 Existing Problems in Version Control

Many version control systems exist that are very successful in solving particular problems. For example, pure version control systems, such as *SSC* and *RCS*[4], use delta techniques to solve storage problems and keep track of changes made by different programmers to a file. Software configuration systems such as *MAKE*[4] allow for automatic configuration of a system when changes are made to a component.

The integration of hierarchical-structured entities and version control is still not satisfactory. Using a system such as RCS and SCCS, for each file, there is a tree of revisions. The truck is considered to be the *main* version, the branches correspond to *variants*. Often, the changes are *merged* back into the trunk. The tree structure does not show how this merge took place. If integrated environments, such as *Adele*[4], are used, then for each module, there can be variants of the specification. For each specification, they can be variants of the implementation. And then for each implementation, there is an RCS-like structure for the development of implementation. In both cased, a tree structure is used for versions, yet, the tree structure is not appropriate for software development, because of the constant merging of different changes to the same system. A direct acyclic graph would be more appropriate. For example, suppose a program is

written to work with a standard screen in English. Two people independently modify the program. The first adds a graphics interface, and the other changes the error message to French. And then someone asks for a version that has both graphics and French messages. The new version inherits from its two ancestors. The main weakness of existing tools is that the different versions of a component have only a local significance. It might be the case for example, that there is third version of component A and also a third version of component B. But there is no a *prori* reason to expect any relationship between the third versions of separate components. This lack of correspondence between versions of different components makes it difficult to build a complete system automatically.

An approach proposed by W. W. Wadge and J. A. Plaice based on IL provides an efficient rule for generating a version [4]. In this approach, variant concept is addressed. It presents the need for versions of complete systems, and informally presents the versions of complete system can be generated by the versions of components. The approach introduced here will give the users the freedom of building any desired combination, but it is the user's responsibility of deciding which of the number of possible combinations will yield a consistent, working instance of the system.

The advantage of the approach is that it is now possible to talk of versions of the complete system—formed, in the simplest case, by uniformly choosing the corresponding versions of the components[4]. Suppose, that we have created a *fast* version of every component of a compiler. Then we build the *fast* version of the compiler by combing the *fast* versions of all the components. Of course, in general it is unrealistic to require a distinct *fast* version of every component. It may be possible to speed up a compiler by altering only a few components, and only these components will have *fast* versions. So we extend our configuration rule as follows: to build the *fast* compiler we take the *fast* version of each component, if it exists; otherwise, we take the ordinary *vanilla* version. As generalized, in this approach, a system is perceived as a hierarchical structure that is composed of components or modules, each of which has its own set of versions. The system is configured with the most relevant versions of each component.

2.3.2 Intensional Solution

The following mechanisms are adopted to configure a particular version of a system.

2.3.2.1 the Intensional Configuration Rule and the Refinement Order

A partially ordered algebra is defined among versions, it reflects their refinement relations. *$V \subseteq W$, read as "V is refined by W", or "V is relevant to W", means that W is the result of further developing version V. The basic concept is that in configuration version W of a system, we can use version V of a particular component if the component does not exist in a more relevant version. That is, we can use version V of the component as long as the component does not exist in version V', with $V \subseteq V' \subseteq W$*[4]. The major advantage of defining a refinement relationship between versions for each module is that the creation of a system version can be automated. Note that the most generic version of each component (*vanilla*) must be created as a last resort in the system in case the system cannot find more refined versions of a component.

2.3.2.2 Subversion

If a piece of software is going to be adopted in different environments, it must meet the different requirements of the users. The differences could exist at many levels. For example, at the level of user interfaces, some prefer text menus while others prefer the visual interface and mouse, while at the functionality level, some clients are provided with full access to all functionality while others only partial access. Different in implementation may also arise from one machine to another. The versions for machines X and Y may be identical, but differ with that for machine Z.

In this approach, *subversion* (variant) is introduced. It is partially addressed in SCCS and RCS, with the introduction of branches, unfortunately, replying on the numeric strings to identify the branches becomes very unwieldy. We choose the path of naming the subversions. For example, a user can define a *bugfix* version of *2.3.4*, and a *release* version of *2.3.4*, a *bluesky* subversion of an *image* version or an *indigo* subversion of the *image* version. Subversions of a version are especially useful for parallel development.

2.3.2.3 Version Join

By joining compatible subversions according to the refinement rule, different system versions can be created. The "+" sign is used to identify the join operation.

The concept of *least upper bound* is addressed in the joining of compatible versions, which refers to the most relevant version of a component according to the refinement relationship specified.

Consider version V1 and V2, version V1+V2 is the least upper bound of V1 and V2, if and only if for all V such that V1 $\subseteq$ *V and V2* $\subseteq$ *V then V1+V2* $\subseteq$ *V exists. In other words, V1 +V2 is the least upper bound (the most relevant) because of the axiom* $(V1+V2) \subseteq (V+V) \equiv V$[4].

As above, we can create a new version for example, *Japanese+ graphics+ infinite X*, if all of the versions of the modules are compatible.

2.3.2.4 Discussion of the Approach

Based on the refinement relationships, a system version can be generated automatically, and the number of possible system versions is greatly increased, because this approach allows users to combine arbitrary versions for each component.

The intensional semantics of this version control approach is that the version universe of the system contains arbitrary user-defined possible worlds. A version of software can be perceived as variables in the universe. It is configured by the most relevant coordinate in each possible world. Using this approach, the version universe is expanded with each increase of user-defined dimensions. This approach was successfully implemented in IHTML, which extends HTML by allowing arbitrary user-defined versions for a site.

2.4 Applications of the Version Control Approach to WWW----Intensional HTML

Based on the TCP/IP protocol implemented in the Internet layer, HTML and HTTP are used between the clients and the server.

On a web site, HTTP is used as the basis for building a true client/server multimedia environment. HTML documents contain links to different data types. Web

browsers use plug-ins to interpret the different types of data and display it on the screen. This multi-media environment can consist of all kinds of information that a computer can process.

HTML documents contain markup codes called tags in their body to control how the text is displayed. The markup tags tell the browser how the marked up text should be displayed—for example a header, link, bullet, list or body text. They can also contain other information such as a URL or the file name of an embedded picture. The format and usage of the different tags are detailed in the following sections.

2.4.1 IHTML Introduction

IHTML introduces a version control mechanism to HTML by providing an efficient representation of the web's intensional characteristics. (See more information for IHTML in reference [3].)

In IHTML, a web site is a hierarchical structure composed of many user-defined modules, each of which has multiple user-defined versions. A particular version of a web page is composed of the corresponding versions of each module.

In THTML, a version is specified with a *dimension name* and a *version value* in that dimension. For example: *bgc:blue* indicates a version is blue at the user-defined dimension: background. Join is represented as a "+". A version of a web site can be *bgc:blue+ lang:english*, which is composed of the most relevant versions of *blue* in the *background* dimension and *english* in the *language* dimension. In all IHTML systems, a vanilla version, named **`aai.html`**, should be defined for each module.

IHTML follows the configuration rules introduced in the above section. IHTML is implemented by keeping only one copy of all variants; the server generates the final system version dynamically using the most appropriate version of each module. After sending the resulting HTML files to the client, the server discards it, so that maintenance of these versions can be avoided.

From the intensional semantic perspective, IHTML provides context-switching operator. This allows users to switch contexts and modify versions of each element on web.

2.4.1.1 Intensional Context Switches in Web Elements

A web element is an entity defined with a special tag and attributes in HTML such as a hyperlink, SSI. In IHTML, because context switch operators are inserted into the web elements, web elements become intensional web elements, each of which has an intensional and extensional expression. The intensional context switches are implemented in hyperlinks, images and links in IHTML.

- **Intensional Context Switches in Hyperlinks**

Extensional expressions in hyperlinks for example, have the following format:

```
<a href= "URL_Address" VERSION="lang:English"> highlighted_text
</a>
```

in which the absolute version of the URL is given no matter what context the current page is in. This means that, for example, if the current page is in **lang:french+pic:big**, then the version of the above hyperlink is modified to **lang:english** defined by the **VERSION** tag.

Intensional context switch in a hyperlink defines the version or dimension modifiers of the link so that the user can switch to the relevant version based on the current one. In the above example, if the **VERSION** is changed to **VMOD**,

```
<a href="URL_address" VMOD="lang:english"> text </a>
```

the version of the URL therefore becomes **lang:english+pic:big,** which is the result of the merge of the current context **lang:french+ pic:big** and the local context modifier on the *lang* dimension, namely **lang:english**.

- **Intensional Context Switches in Image Elements**

Context switching operators are embedded in an image element in the same way as hyperlink. The extensional and intensional expressions of an image are distinguished with **VMOD** and **VERSION** respectively. For example:

```
< img src="image.gif" VMOD="lang:french"> French version </a>,
```

which defines the version of the **image.gif** based on the current version of the page, while

```
<img src="image.gif" VERSION="lang:french"> French version </a>,
```

which defines the absolute version of the **image.gif** to **lang:french**.

- **Intensional Context Switches in Server Side Includes (SSI)**

IHTML also has a *Server-Side Includes* feature that causes the contents of the included file to be incorporated into HTML page at the server before it is sent back to the client.

The advantage of it is that the included information is included on the fly at request time.

A SSI may have multiple versions, the files included in server's response will be the best-fit version for the user's request. For example

```
<!--#include virtual="header.html" -- >
```

if the page's version is currently French, the file included will be **french** version of

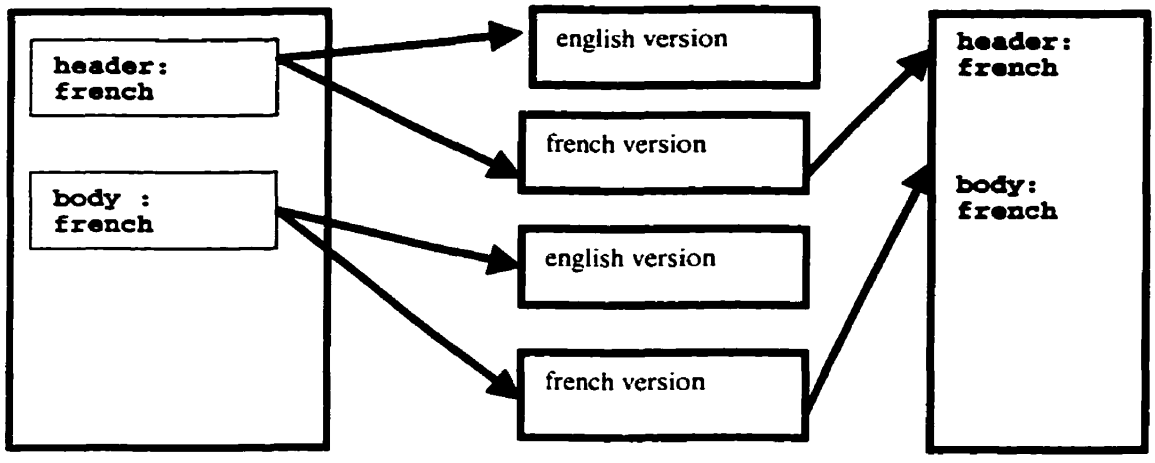


Figure 2.1 IHTML Working Model

header.html if it is in storage; otherwise, the vanilla version will be sent instead. (An error message will be sent to the user if the *vanilla* version of the target file cannot be found.)

2.4.1.2 Multiple Candidates and the Best-fit

The IHTML system keeps multiple versions of each web module. When the server receives an IHTML file from a client, the file pools are searched for the one matching the client's request. If more than one fits the request, the best-fit version will be found according to the ambiguity-handling algorithm. Figure2.1 illustrates the IHTML working mechanism.

2.4.2 Summary of IHTML

As an IP language on WWW, IHTML integrates intensional semantics into the HTML and context-switching operators for the user to switch among different contexts. There are many advantages of IHTML over HTML.

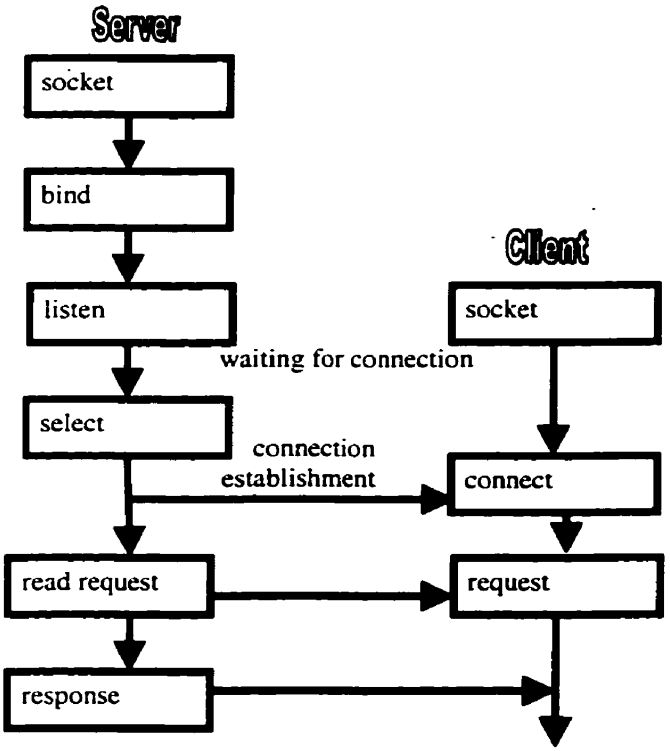


Figure 2.2 Typical Client/Server Model

First, from functionality perspective, it provides a natural method to support arbitrary user-defined versions for a site, the user can switch versions for an element or a whole page by context switching operators, and the version of each can be totally or partially modified. IHTML provides great flexibility to the user in version

modifications.

Second, in the implementation, it saves system storage greatly by keeping versions of variants rather than the system versions, in which there may be cloning parts among them. The server dynamically creates a page by the configuration rule of the system version mentioned in Section 2.3. The version configuration of a site is automated to the greatest extent.

Third, it provides a base from implementation of THTML, which will be presented from the next chapter.

2.5 Base of THTML 1.0

The software of THTML is implemented on APACHE server API. In order to introduce the implementation of the THTML system, HTTP protocols, which the HTML protocols and the servers' behavior are based are introduced first as the background knowledge.

2.5.1 HTTP Protocol

HTTP protocol is based on TCP/IP. The typical web model is client/server model, in which, the client initiates a TCP connection with the server, after the connection is established, the client sends a request down that channel. The server examines the request and responds in a manner specified in the server *Apache API* and third-party modules. When a user types in a URL, *the protocol name, (http in our case), server's name(www.somewhere.com), the directory name(/where/foo.html)* and the *port number (use the default if not specified, 80)* are used to establish a unique connection with the server. The request for an HTML document is issued to the server on the dedicated network with the following format:

```
GET /where/foo.html HTTP/1.0<CR><LF><CR><LF>
```

GET defines the method of retrieving a resource (normally a file) from the remote host. **`/some/where/foo.html`** defines the virtual host and the directory name, i.e. where the file is located. **`HTTP/1.0`** is the protocol name. The server sends the file

back if it can find it on the server or decline the request if the resource does not exist. If no more requests happen during a certain time, it will close the connection to the client.

2.5.2 Server-Side Software—*Apache Server API_1.2.5* [17]

In THTML, special actions are defined at the server to handle THTML requests from the web clients. The software is implemented by extending a module based on Apache API server software, which offers a high degree of drop-in compatibility with the popular NCSA server [2]. Internally, an APACHE server is built around an API (application programmer interface) that allows third parties to add new server functionality. Most of the server's visible features (logging, authentication, access control, CGI, and so forth) are implemented as one or several modules. The implementation of request/response handling is discussed in the following sections.

2.5.2.1 Common Server Behavior

At the back-end of the server, after the server software has been compiled and got running, a daemon process will be running in the background, listening to any requests for connection. After one is detected, it forks a child process to handle it. Based on the data structure received from the client, it will determine which host and port number are the targets of the request. The virtual host then takes the path inserted in the request, and reads against its configuration to decide on the appropriate response.

2.5.2.2 Apache Solution Features

Apache makes the server running with default behaviors or customized behaviors defined in users' configurations and third party modules. There are two factors determining the server's request/response, namely the *directives* in configuration files and the *extended modules* at the server.

- **Directives**

Directives are provided in the configuration files to allow a user to specify his/her preference on server's behavior. For example, *a port number(8080)* and the *server name(valdes.uvic.ca)* have to be defined as well as a *log file* for THTML for recording

the behavior of the server. These features are all conveniently specified in corresponding directives in a configuration file.

- **Module Structure**

Apache handles the requests and response by several steps (phases). They are:

1. URI to filename translation
2. Several phases involved with access control
3. Determining the MIME type of the requested entity
4. Actually sending data back to the client
5. Logging the request.

The user can extend modules to handle any or all the phases. Some phases go through all the modules linked with it before going to the next phase such as logging the request and handling access control, while others may stop if one of the modules linked with the step successfully handles the request such as the step of URI to file translations, in which if one module handles the request and successfully translates the URI to a file, it will stop and go to the next step.

In the THTML implementation, a module is extended to handle the translation of a temporal URL and sends clients a response. The implementation of THTML is presented extensively in Chapter 5.

Chapter 3

Time in THTML

3.1 Overview

There are three components that distinguish THTML from conventional HTML for both the reader and author of web sites.

1. Time sensitive tags are used to allow a reader's browser to specify a request time. This request time specifies which instance of a page is desired.

2. The author can create multiple interval-labelled source files for a single page or component. These interval-labelled files indicate when a particular page should be posted.

3 The times in these pages and requests can either be relative or absolute representation.

Some questions still remain, however, on how the above will be implemented. How will time be specified such that it reflects the temporal requests from readers and authors? There is also the question of how a request with different possible matches is resolved. These questions will be answered in the following chapter where we will introduce the existing temporal models and their advantages and disadvantages. We will also discuss a time-interval refinement algorithm that will help to resolve any ambiguities resulting from different possible matches.

3.2 Introduction to the Temporal Interval Stamp and the Time Sensitive Tag

Two new components were incorporated into THTML to support temporal requests. They are time sensitive tags (TST) and time interval stamps (TIS). TST's are used to specify which instance of a particular page should be sent to the client. For example, if we wanted an image of the weather outside at some specific time, we would specify the exact time in the TST and it would be the server's responsibility to match that tag to a

particular image.

On the server side, it uses the TIS associated with each HTML file to determine which instance should be sent to the client. The server is responsible for determining the best-fit HTML page for each temporal request and sending the results back to the client. How the server determines the best-fit page and the design of both the TST's and TIS's will be discussed further in the following sections.

3.3 TIS Design

3.3.1 State-of-the-art Time Representations

Although time is continuous in nature, two common views have evolved: *continuous* versus *discrete* time. Continuous time is considered to be isomorphic to *real numbers*, whereas discrete time is normally isomorphic to *natural numbers*. The discrete interpretation of time has been adopted in many real-time systems because of its simplicity and the relative ease of implementation.

There are two time models widely adopted. One is the *linear time model*; the other is the *branching time model*. In the *linear time model*, time is considered to be totally ordered, i.e. if two distinct time points t and t' are given, either t is before t' or t' is before t chronologically. However in the *branching time model*, multiple time lines are defined and a chronological ordering only exists between time points on the same time line. The branching time model is often adopted in hypothetical reasoning, where time in the future cannot be determined, therefore, multiple time lines are assumed and researched hypothetically. For simplicity, we only consider the linear time model in this thesis. Therefore, time points are totally ordered on this single time line. Nevertheless, it is possible for web designers to use a branching time model, depending on the application.

3.3.2 Time-point and Time-interval Representations in THTML

The time model is not the only issue that needs to be resolved, there has also been a lot of debate on the most appropriate representation of time. The two most common

representations are the *time interval* and the *time point*. An obvious choice is the time point. One reason for this choice is its simplicity. Time points are isomorphic to numbers and therefore easy to represent. The other reason is that the computational complexity of dealing with time points is less than that of time intervals.

The main drawback of the time point representation is that it is cumbersome for expressing the fact that an entity holds over a time period. As long as time is discrete, every time span can be represented by a finite set of time points. While this argument is theoretically correct, it is often not practical. For example, to specify that a THTML file holds for a period of time on Web, the time period would consist of a list of all valid time points. This representation is especially cumbersome when the granularity of time is fine-grained (e.g. seconds or even finer) and the average time a file holds is relatively long (say months or years). An obvious way out of this dilemma is to use time intervals in order to capture the duration of a temporal file. In addition to modeling the time span, the time interval can also be used to model time points if the start and end points are the same.

The main drawback of the time interval representation is that the domain of the intervals is not closed with respective to the usual set-theoretic operations (union, difference and intersection). For example, the union of the two intervals may yield two intervals or just a single one.

$$[4\text{-}10] \cup [15\ 20] = \{[4\text{-}10], [15,20]\}$$
$$[4,10] \cup [6\text{-}17] = \{[4\text{-}17]\}$$

The behavior of subtraction (set difference) is even worse:

$$[4\text{-}10] \setminus [8\text{-}20] = \{[4\text{-}8]\}$$
$$[4\text{-}10] \setminus [6\text{-}8] = \{[4\text{-}6], [8\text{-}10]\}$$
$$[4\text{-}10] \setminus [1\text{-}20] = \{\}$$

Depending on the operands, subtraction yields zero, one or two intervals as a result.

This property makes it quite awkward and expensive to process time intervals on the computer. [14] Therefore, a question arises--namely whether we should use the *point-based* model or the *interval-based* model in THTML.

Because it is common for people to specify time intervals over which files are to be posted on web, the time interval representation is needed to represent the valid times of files. In order to avoid the complexity it may induce, THTML adopts the representation of a *union* of sets of time intervals rather than *difference* in order to simplify the implementation. Time intervals are perceived as the shorthand of time points within the starting and ending points inclusively. They are both adopted in the definition of TIS. In THTML, a reader must supply a complete time point to retrieve an instance. THTML doesn't support *request intervals*, therefore, in the TST specification, only time points are adopted. However, both time intervals and time points can be defined in the time-interval stamp.

3.3.3 Time Pattern Design in Time-Interval Stamp

3.3.3.1 Temporal Data Type in THTML

The data type of the TIS should have the capability of presenting both a time interval and a time point. Before discussing the design of the data type, we first clarify a terminological problem. There are two terms widely used to represent time: *date* and *time*. Sometimes, the distinction between a date and time is made based on the chosen granularity. A date covers the granularity *year, month* and *day* while time covers every time unit below a day. Therefore, for example *1985/3/25* is called a *date* while 13:45 is called a time. Depending on the chosen granularity, it may be more appropriate to use one term over the other.

Nevertheless, THTML combines the two as the time type, which is *time + date*. There are two main reasons for adopting this data type. First, an acceptable granularity for representing the valid times of a file needs to be provided. Normally, on a computer, the granularity of files is in seconds. Although there are some arguments that the development of files should be granulated to below a second, the speed of web transfers

is insensitive to precision below a second. So the granularity in THTML is a second. This includes the modified time, the expired time, and the user-request time. This corresponds nicely with most version control software that also records time with precision of a second. The second reason is portability. The THTML time-keeping mechanism is consistent with that of all computer systems in use today—Unix, Windows, MAC and DOS. (whose time granularities are all up to seconds) So THTML is not tied to any specific architecture or operating system.

The time type is specified using *six* units. The format is as follows:

```
second minute hour day month year [dow]
```

The valid range of each unit is the same as that in the time-keeping mechanism of Unix as follows:

second [00-59]
minute [00-59]
hour [00-23]
day [00-30/31/29/28]
month [1-12]
year [0-]
dow (days of week) [0-6]

Legend:

Among the seven units, *dow* (day of the week) is optional, it is provided to facilitate the temporal specification. There are two reasons that the unit of *dow* is added. First, in some cases, people have a habit of specifying a time pattern based on the day of week. Second, if there is not sufficient information given by the preceding time units to calculate the date, the server calculates the date from the data in *dow*. However, if there is a conflict between the *dow* and a specific date of the month, the value in *dow* is ignored.

Some users have a habit of specifying some of the time units while ignoring others, so, • is specified to indicate arbitrary values in the valid range of a data unit.

3.3.3.2 Time Pattern Definition

There are five basic types that are supported in temporal logic: *during*, *after*, *before*, *point* and *recurring*. Especially, the *recurring* pattern facilitates the web authors to specify web postings that occur periodically.

The precise definition and their representations are as follows:

During, *after/before* and *recurring* identify time interval patterns; whereas *point* indicates a time point pattern on a single time line.

During is a bounded interval specification. It occupies only one segment on the time line. "-" is used to specify the interval between two points. *After/Before* are unbounded intervals. "<" or ">" preceding a time point is used to denote the time period *after* and *before* a particular time point inclusive. *Point* occupies only one point on the time line. *Recurring* describes multiple time-points or bounded segments that appear periodically on the time line. It is provided especially for sites that change contexts frequently, such as commercial sites, "," is used to specify a recurring time pattern. So for example, the time-interval stamp

```
> 10 30 52 1 2 1998 *
```

indicates a file should be posted *after* time point 10:30:52 February 1 1998 while

```
< 10 30 52 1 2 1998 *
```

indicates a file should be posted *before* time point 10:30:52 February 1 1998. If a web author intends to post a file on every Monday in January, February, March 1998, he can write the following time tag in the TIS:

```
* * * 1,2,3 1998, 1
```

If he determines to set the time period from Jan to March 1998 to post a file, he can write in the TIS:

```
* * * * 1-3 1998, *
```

One can combine *recurring* patterns with *interval* patterns in usage. Note that with the "-" in a row, the user can specify the *starting* and *ending* points of an interval on the time line. For example, time-interval stamp

```
* 1-2 3-4 5-6 1998 1,2,3
```

indicates a file should be posted every *Monday*, *Tuesday*, and *Wednesday* in the time

period from *May 3rd 1998, 1am* to *Jun 4th 2am 1998*

3.3.3.3 Discussion of Normalization Operation

A normalized time pattern is one that consists of only the union of a set of time intervals. The process of converting all patterns to this form is called *normalization*. It has great significance in judging tractability and is also an efficient method for matching temporal requests with time patterns

Since *difference* operations are not defined at the moment, normalization is not needed in THTML. However, *difference* operations will be included in the future, and so all time patterns must be normalized so that the same matching algorithm can be used without any modifications.

3.3.4 Conclusion of Time Pattern Design

Although there are many data models that can be used to represent time, we have chosen a model that is able to support most common temporal requests and is also easy to implement. But the question arises, is it the best representation for a temporal language? Is there a measure for evaluating the expressive power of a temporal language? If yes, is it possible to have a canonical expression independent of various time representations? There are many debates on this topic, so far, there is no answer in the literature.

Nevertheless, THTML can still be improved in the following aspects:

1. One further direction is incorporating more complex logic into the system so that the expressive power of simple time patterns can be enhanced. Intensional rules, including mathematical algorithms, can be adopted to specify more complex temporal requirements from the web authors.

2. Another direction is to incorporate a data flow language (temporal programming language) that describes rules for temporal data flow. For example, *every other day* can be expressed naturally and efficiently in dataflow programming language with a definition of time:

```
n= k fby n+2d; k= $Fri May 3 1998
```

Precisely, an infinite temporal stream of dates, namely every other day, starting from May 3, 1998 is defined. The backend support for it includes algorithms for parsing and for temporal rules.

3.4 Best-fit Interval Selection in a TIS

The definition of an interval refinement relationship is the principal design requirement in THTML, since time plays a key role in retrieving documents and ambiguity may occur if the time point in request lies in multiple time patterns in a TIS.

A TIS that brackets the specified time point is a *candidate*. If a single candidate is found, it is used to generate a conventional HTML page. If no candidates are found, the absolute source file is used. If multiple candidates exist, the best-fit candidate is chosen. The rule of retrieving the best-fit pattern is determined by *extensional* and *intensional* rules. With DTL (Discrete Temporal Logic) applied in THTML, an interval can be perceived as a set of discrete time points on the time line, then the basic rule (extensional rule) to choose the best-fit candidate becomes *the smallest subset encapsulating a given time point is the best fit*.

Since not all candidates can be perceived as subsets of another, the extensional rule does not eliminate all ambiguities between time interval candidates. For that reason, we also provide an intensional rule.

3.4.1 Extensional Rule

THTML specifies that the request time in a TST is a time point while the TIS can be either time points or any combination of time intervals. When a time request is received, the server searches for the best-fit interval. The best-fit interval is defined to be the smallest interval encapsulating a given time point. For example, the two time patterns A and B below are both candidates for a time point P, because P lies in the intervals of both A and B.

B refines A extensionally because the set of points in B is a subset of the set of the points in A.

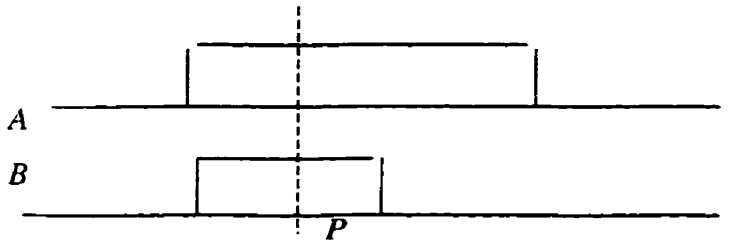


Figure 3.1 Comparison between the Temporal Patterns of the Same Type

- The comparison of "*after*" and "*before*" patterns is also simple:

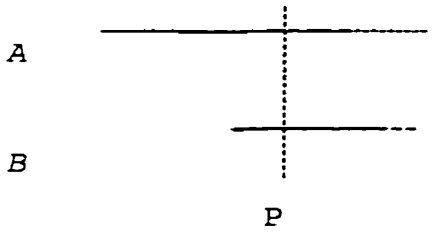


Figure 3.2 Comparison between "after" Patterns

Obviously, in Figure 3.2, the set of points in *B* is the subset of that in *A* to the time criterion *P*, therefore, B is the better-fit candidate. As the same token, in the comparison between two *before* patterns, the one that is the subset of the other is the better-fit.

The absolute time interval refers to the unbounded interval from negative infinity to positive infinity. A conventional HTML file is perceived as a THTML file labeled with the absolute time interval. Using the extensional rule, it is easy to see the *absolute time interval* is the most generic one among all candidates; for it subsets all time patterns. So, the HTML file is the most generic source among all THTML file candidates. It is used as the best-fit when ambiguity happens.

3.4.2 Intensional Rule

The intensional rule applies when comparing as subsets is not enough. It is designed to minimize ambiguities that could result from the extensional rule. For example, in the situation in Figure 3.3, an ambiguity exists because neither of them is a subset of the other.

Ambiguities exist in the comparison of the same time patterns as well as different time patterns using the extensional rule. Let us first discuss all possible combinations of patterns and then we will show how the intensional rule resolves these ambiguities.

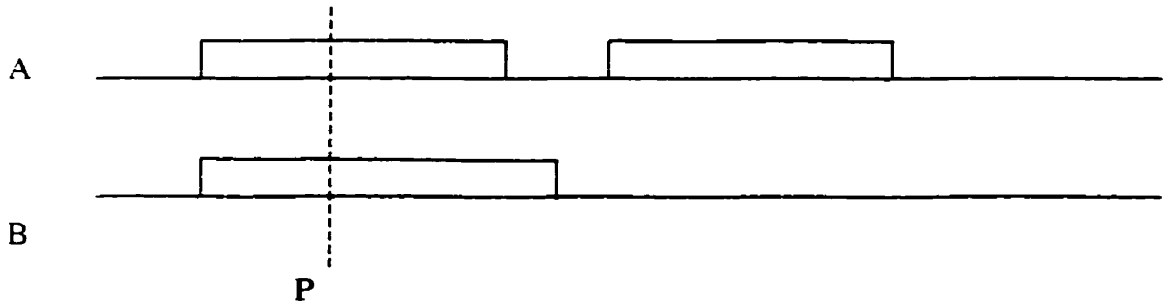


Figure 3.3 Comparison between Different Type of Temporal Patterns

3.4.2.1 Ambiguity Resolution among the Same type of Temporal Patterns

• Since the time tags in TIS are unique, no ambiguity can exist if the TIS consists of just a time point.

• When two infinite *recurring* patterns are compared, ambiguities may occur if one is not a subset of the other. For example, in Figure 3.4, whether *A* or *B* should be chosen as the best-fit is hard to judge.

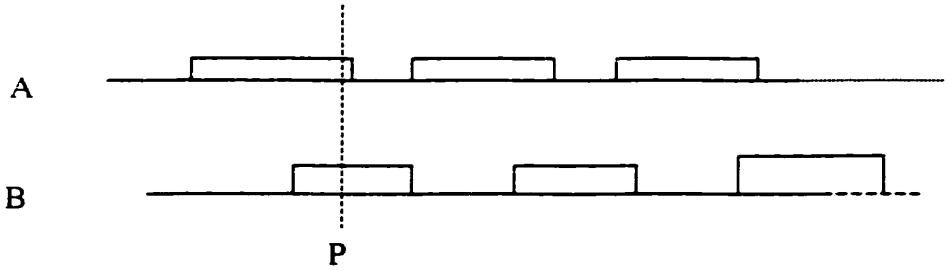


Figure 3.4 Comparison between "recurring" Patterns

To eliminate the ambiguity in this situation, the intensional rule specifies that if the recurring pattern contains equal sized segments, the candidate with smaller segment is the best-fit. The ambiguity occurs in other situations.

• In the comparison between two *during* patterns, extensional rule applies when one bounded interval is a subset of the other. However, ambiguity occurs if one is not a subset of the other. For example, in the following scenario:

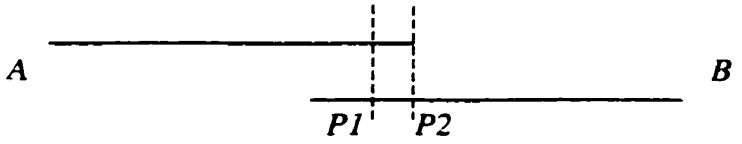


Figure 3.5 Comparison between bounded "Interval" Patterns

If *A* and *B* occupy the same length in the time line, ambiguity occurs with request time P1 or P2.

3.4.2.2 Ambiguity Resolution among Different Temporal Patterns

- The comparison of different time patterns is more complicated than for the same time patterns. For example, it is hard to tell if a *during* pattern candidate is more refined than a *recurring* pattern candidate, therefore, additional refinement rules are needed in the intensional rule.
- We can deduce that the *during* pattern has a smaller temporal gap than the *before/after* pattern, because the *before/after* pattern indicates a time period with an infinite end. This has a larger gap than the finite interval in a *during* pattern, therefore, it is more generic in definition than the *during* pattern.
- The time *point* has smaller time gap than any other patterns, therefore, it is more refined definition than other patterns.
- However, there are some situations where it is hard to find which pattern is more generic in relationship to a particular time point if two different patterns are given, say, a *before* pattern versus an *after* pattern and a *recurring* pattern versus a *during* pattern.

Ambiguity occurs in the first situation because the comparison cannot be solved mathematically between the two infinite segments; while in the second scenario, one is an infinite time pattern and the other is finite time pattern. An infinite pattern is not necessarily less refined than a finite pattern because there may be some situations for which one of the segments of the *recurring* pattern is closer than the *interval* pattern to the time point.

The *recurring* pattern describes a file to be posted on the web periodically. The intensional rule defines that the *recurring* pattern is the *least-refined* among all patterns.

One reason why the recurring pattern is the least refined is because it can be

represented by a set of *during* patterns. The user can just use this method if he/she wants to obtain a higher priority.

The *during* pattern is defined to be more refined than the *before/after* pattern. If instead the *during* pattern were less refined, one could not use any other alternative time patterns to give *during* a higher priority. (Note that we do not allow an interval to be denoted as time points. Although it is feasible in theory, it is not practical to define a large time interval with time points.) There is an example to illustrate the above reasoning. If two time patterns are given, one is

```
> 10 10 10 2 3 4 1988 *.
```

the other is

```
* * * 1,2,3 1988 *
```

If one intends to give a file, associated with the second pattern, a higher priority than a file associated with the first pattern in case that both can be candidates to a request time, he/she can specify the second time pattern as a *during* pattern.

```
* * * * 1 1988 *
* * * * 2 1988 *
* * * * 3 1988 *
```

As long as the frequency of the file is not very high, the above method is feasible.

Consequently, the refinement order in the intensional rule is as follows. From the most generic to the least:

recurring before/after during point

Note that the above order is specified in terms of the time interval patterns adopted in THTML, it may not applicable to other temporal presentations.

3.5 Best-fit Interval Selection among Multiple TIS's

Because multiple time tags are allowed in a single TIS, confusion may occur to the *best-fit* interval among the THTML files. In THTML 1.0, it is not the result of comparison among all patterns in all THTML files at once, but among the best-fits of each file. Precisely, suppose that *P1*, *P2*, *P3* are time patterns in THTML file *T1* while

Q1 and *Q2* are the time patterns in THTML file *T2*. Suppose also that the candidate (relevent) labels are *P1*, *P2*, *Q1* and *Q2* are all candidates to a given request time *R1*. The process of retrieving the best-fit is as follows:

First, the best-fit among *P1* and *P2* in T1 file and the best-fit among *Q1* and *Q2* in T2 are resolved respectively. The best-fit of each is, say, *P'* and *Q'* respectively.

Second, *P'* and *Q'* are compared. The result is the best-fit to the client's request time R1. The THTML file that associates with it is used to generate the instance. The extensional and intensional rules are adopted in both steps.

The result is the same as that of comparing once. The reason for two comparisons twice rather than one is to simplify the implementation. The algorithm is similar to *bubble sorting*, the smaller subset of the first two will be kept to compare with the rest. The more refined in each comparison is always kept to compare with the next pattern. If ambiguity occurs, the most refined is still kept to compare with the rest. If no other patterns are more refined than it, the absolute TIS will be adopted, otherwise, the smallest subset pattern will be adopted as the best-fit.

3.6 Conclusion

The design and reasoning behind the TIS and the rules to retrieve the best-fit intervals were covered in this section. In order to simplify operations involving the TIS, a linear time model was adopted. The model supports both a time interval and time point representation. This was done so that it would be consistent with the different ways users specify time. Since stating time in terms of a date and time of day is more complicated to handle, the two elements are combined together into a time data type, which granulates time in seconds.

This time data type forms the basis for specifying time requests and defining time intervals. Time points are matched to time intervals using the extensional rule. This rule is based on DTL, which the time interval is perceived as a set of time points on the time line. The time interval that is the smallest subset among those encapsulating the time point is the best-fit. This rule however still leaves some ambiguities. Most of these ambiguities can be resolved through the use of the intensional rules, which define a

refinement order for the different time patterns. The application of these two rules in one file or among best-fit patterns of multiple THTML files allows a server to determine which source file to instantiate given a URL request.

Chapter 4

THTML System Design

4.1 Overview

The THTML system can be perceived as a typical three-tier architecture. It consists of the *clients' browser*, *file system* and *web server*. Enhancements are adopted in all three of these tiers. From the clients' point of view, two features (TIS and TST) are added to allow the client to specify a time element. The server supports this enhancement by implementing a tree structure and adopting a naming convention that maintains the hierarchical relationship between the modules and the different temporal versions of a module. The server determines which page to return to the reader by using a *temporal versioning* and *time irregularity handling algorithm* to generate the most relevant instance for a particular URL request. The THTML system can be illustrated by the communication among the three components (See Figure 4.1).

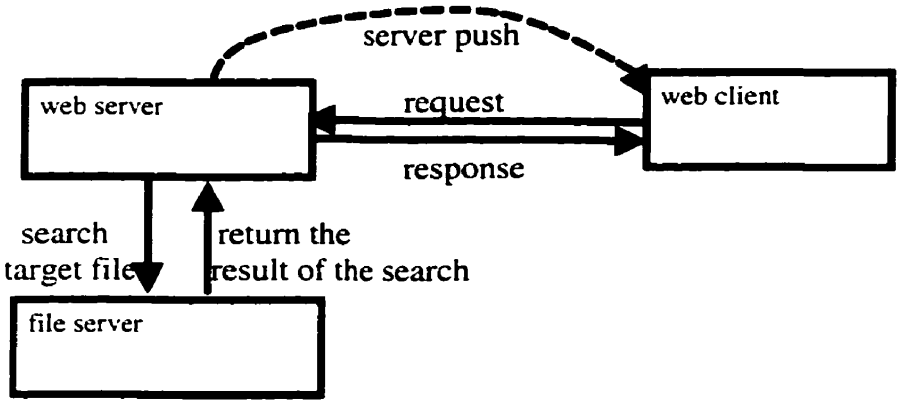


Figure 4.1 Three-tier Model of THTML System

Upon receiving a URL request, the server decodes the address and extracts the request time. The server then searches the file system for the best-fit THTML source file

for that request time. It will then generate a conventional HTML file in response. In addition to the enhancements to the conventional web model, in which the server passively responds to the clients' requests, the *server push* (related to client pull) technology has been incorporated into the THTML server. This allows the THTML web server to actively feed clients with the most updated information at user-defined intervals.

In the following sections, we will discuss the issues related to implementing the above features. First the language design will be introduced in Section 4.2. In Section 4.3, the backend system design is discussed, followed by the algorithm design in Section 4.4. The chapter will conclude with possible future directions for each of the designs.

4.2 Language Design

4.2.1 Design Requirement

In THTML, a web page is assembled using components indicated in a THTML page. These components are single-versioned entities in conventional HTML, but have multiple variants in THTML. Semantically, THTML allows a conventional HTML page or its components to have multiple instances corresponding to the different time points. The generation of a URL instance at a specific time is, in the simplest case, a point-wise operation. It is generated by collating an instance of each component evaluated at the specified time.

The following figure shows the hierarchy in a THTML system. Each web page contains a list of components (images, sounds etc.) to be included in the page. Under each of these components, there are multiple instances corresponding to the different time intervals during which that component should be displayed.

Since THTML allows multiple instances for each URL, there must exist a way of determining which instance the user wishes to retrieve. This requires enhancements to the syntax and changes to the semantics of the HTML language. The syntax is enhanced by adding TIS's to the source files and allowing users to specify time points along with their URL requests. On the semantic side, it involves interpreting temporal context

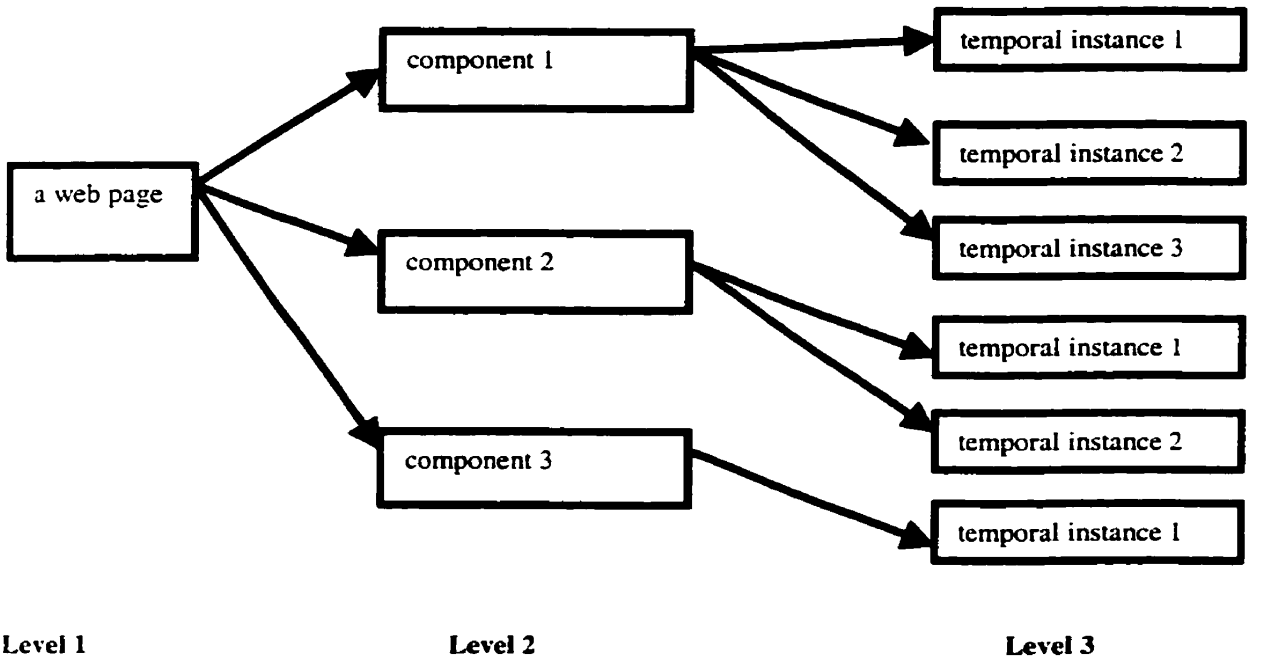


Figure 4.2 THTML Enhancement into HTML

switching operators and helping to evaluate the URL at a particular time context and generating a response.

4.2.2 THTML Language Enhancements to HTML

As mentioned in Chapter 3, a THTML file differs from an HTML file in two ways, namely, the addition of TST's and TIS's. A THTML file has the following format:

```
<cron> temporal_interval_patterns </cron>
<html>
<title>
.....
</title>
<body>
THTML body
</body>
</html>
```

The TIS is inserted at the beginning of a conventional HTML file with a special pair of tags **<cron>** and **</cron>**. The THTML body is similar to a conventional HTML page except that it can contain optional TST's.

THTML body : : = *THTML components*

THTML components : : = *(conventional HTML components)* | *(conventional HTML components having time sensitive attributes)*

THTML does not add any new tags in THTML body, however it modifies the existing tags by adding a new optional attribute. A **TMOD** attribute is added to the most common tags, found in HTML files, to allow them to specify a temporal context switch. The TST's currently implemented are conventional hyperlinks, images and SSI's. This attribute can be added to other tags in the future. There is no limit to the number or sequence of TST's in a THTML source file.

4.2.2.1 Time Sensitive Tag (TST)

Theoretically, any web component can be *temporalized* (made to vary in time). For example, temporal applets and images can be created whose response to the reader is dependent on a time variable. In THTML 1.0, three components were made temporal. They are *images*, *hyperlinks* and *SSI's*. The BNF format of the *TST's* for these web components is specified as follows:

Time-sensitive-tag (TST) : : = **[<!--#include virtual="name.html" TMOD=***temporal_context_switch* **> <src img="imgfile" TMOD =** *temporal_context_switch* | **<a href= "my filename.html" TMOD=** *temporal_context_switch* **>]***

- The syntax of the *temporal-context-switch (TCS)* variable defined in BNF format is as follows:

TCS : : = " " " *temporal_term* " " " " ;

temporal_term ::= *temporal_base* |["+" |"-"] *n* ("**Y**" | "**y**") | ["+" |"- "] *n* ("**M**"|"**m**") | ["+"| "-"] *n* ("**D**" | "**d**")|["+"| "-"]("**H**"| "**h**") |("+"| "-") *n* ("**I**" |"**i**")|["+" |"-"] *n* ("**S**"| "**s**") ["+"| " -"] ("**W**"|"**w**")

n::=*digits*;

temporal_base ::=ε| *natural_l* | *absolute_t;*

natural_l ::= "**tomorrow**" | "**yesterday**" | "**now**";

absolute_t ::= "**$**" *Week_day Mon Date Hour* "**:**" *Min* "**:**" *Sec* ;

Week_day ::= "**Mon**"| "**Tue**"| "**Wed**"| "**Thur**"| "**Fri**"| "**Sat**"|"**Sun**";

Mon ::= "**Jan**"| "**Feb**"| "**Mar**"| "**Apr**"| "**May**"| "**Jun**"| "**Jul**"| "**Aug**"| "**Sep**"| "**Oct**"| "**Nov**"| "**Dec**";

Date::=*digits from 0 to 31;*

Hour::=*digits from 0 to 23;*

Min::=*digits from 0 to 59;*

Sec::=*digits from 0 to 59;*

Legend:

• the time unit identifiers: **Y, M, D, H, I, S** (capitalized or lower case) indicate temporal units: *year, month, day, hour, minute and second,* respectively.

• As shown above, simple temporal terminology can also used in context switch attributes. The terms **yesterday**, **tomorrow** and **now** are supported in the current version and their semantics will be presented in 4.2.3.

4.2.2.2 Time Interval Stamp (TIS)

The TIS is the other feature added in the conventional HTML. TIS's are distinguished from other components by a pair of tags **<cron>** and **</cron>**. The user can specify a specific time point, a time period or a set of time points and periods. All TIS's can be specified with six time units as follows:

```
second  minute hour day  month  year [dow]
```

Each unit is separated with white space. The days of the week (*dow)* is optional. Each unit can be specified using a combination of numbers and symbols. The following

symbols can also be used:

- denotes a time interval.

, separates a list of valid inputs.

* denotes any arbitrary value in the valid range of a data unit.

Therefore, for example, the time interval from May 3, 1998 to May 20, 1998 can be represented as follows:

```
<cron> * * * 3-20  5 1998 </cron>
```

4.2.2.3 Creation of a THTML file

THTML 1.0 provides a command in the UNIX environment for an author to create and modify a THTML source file. The command **ivi** originates from IHTML 2.0 and was used to create an IHTML file (See [10] for more details.) The author enhanced the command to create a THTML file. The format is as follows:

ivi [-t] [-*<number>***]** *filename*.**html**

The command allows the user to edit the specified THTML source file. When it is invoked, it will open the default editor. (specified by the $EDITOR environment variable) The **-t** option is used to create a new version of *filename*.**html** using a copy of the most recently created version. This new version will be saved in the *filename*.**html** directory in the following format: **aai.***n*.**html**, where *n* is a unique number generated automatically by the system for each version. An existing version can also be edited by specifying the version number in place of *<number>*.

4.2.3 Language Semantics

Formally, if each conventional HTML page is P and the request time is u, then each THTML page can be specified by (P, u). At request time u, a unique instance of P is issued to the client. From the semantic point of view, the response to a URL request is an HTML page that is dependent on the time point specified. In other words, the URL is an intension. The response to the reader corresponds to the process of generating an extension from the intension. An extension to a particular intension, namely an instance is generated using two components. One is the URL or HTML component, the other is the request time. The temporal contexts are not necessarily the client's request time. The

reader can define any time point. The web elements in the response for the URL inherit the context by default. However, they can also be modified locally by temporal context switches.

4.2.3.1 Semantic Distinction between HTML and THTML

4.2.3.1.1 The Global Context and the Local Context

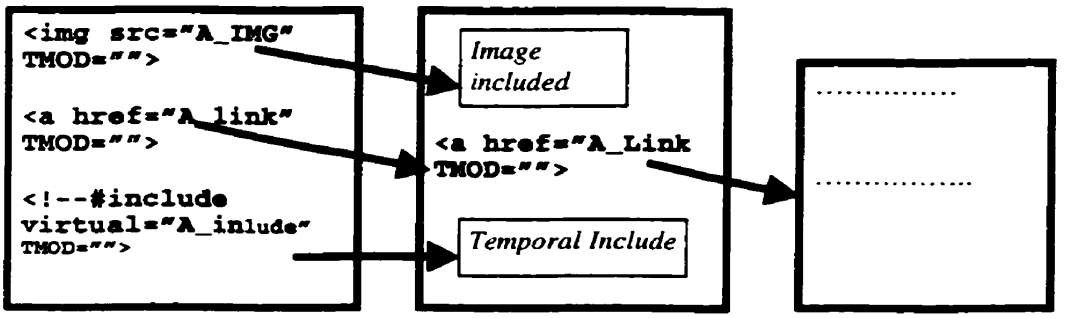


Figure 4.3 Server Interpretation Model

The *Global Context* and *Local Context* are distinguished by their scope. Because the context for the URL is adopted as default to interpret those for the web elements for the HTML page. In other words, the elements in the HTML page inherit the context of the URL by default. However, a context switch operator can be defined in each local element to modify the global context. It changes the six time units as necessary.

At the server, a THTML document is converted into conventional HTML. Each component is instantiated using its local context. Figure 4.4 illustrates the conversion process.

4.2.3.1.2 Temporal Context-Switches (TCS)

Global contexts are converted to local ones by *temporal context switches*. These temporal context switches are implemented only in *hyperlinks*, *images* and *SSI's* in THTML 1.0. The usage and semantics of them are covered as follows:

Temporal Context Switches in Hyperlinks

A temporal context switch can be specified using the attribute **TMOD** in a hyperlink:

```
<a href= "A LINK" TMOD=TCS>
```

When the hyperlink is invoked, a conventional HTML file, for the URL, is instantiated using the context specified in *TCS*. The local context of a link containing a

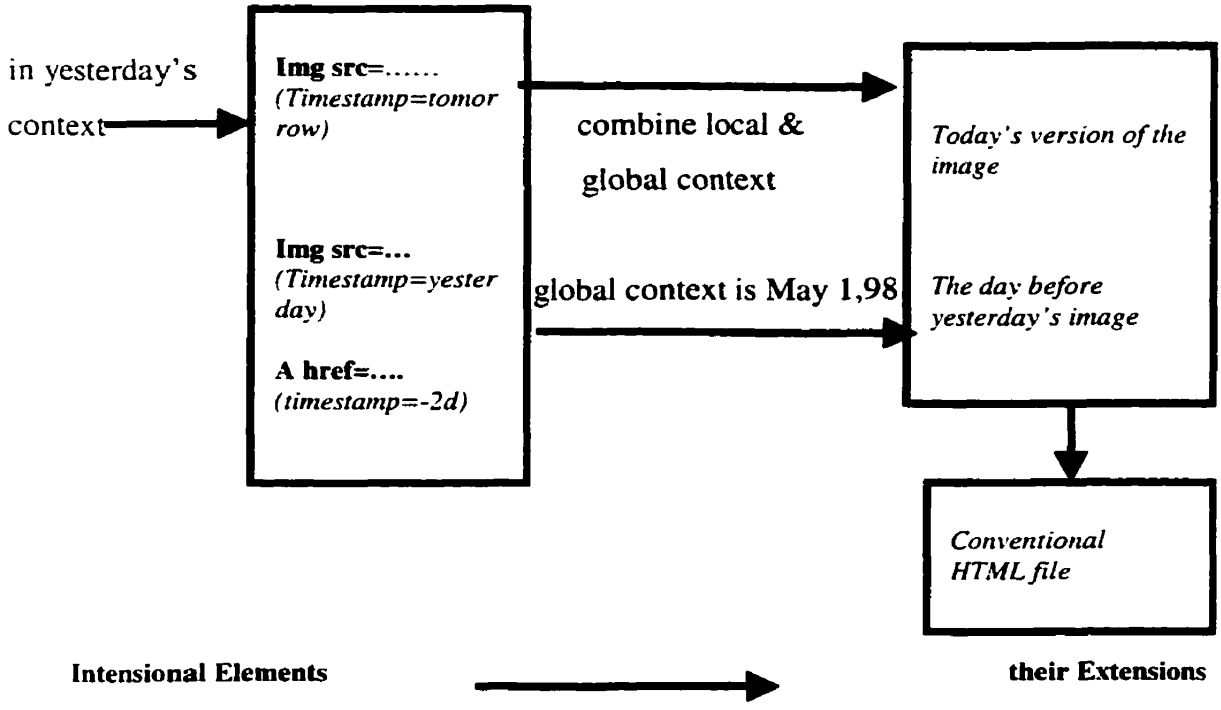


Figure 4.4 the Local and Global Context

TCS is a combination of the base temporal context and the local temporal modifier. If a base temporal context is not specified, the global context of the current page is used.

The temporal words **tomorrow**, **yesterday** and **now** are implemented in THTML as following manner:

- **Tomorrow** is equivalent to the local modifier "**+ 1d**". In other words, it represents the same time (the same hour, minute and second) as the default context, but increments the day by 1.
- **Yesterday** however has the opposite effect and decrements the day by 1.
- The keyword **now** however denotes *the default global context.*

These three temporal terms are interpreted in the same manner in the other elements.

Extensional expressions can also be specified in TCS's. For example, **<a href="URL_Address" TMOD="$Tue May 19 12:13:14+2H-1D">** denotes that the local context is changed to *May 18 10:13:14.*

Temporal Context Switches in Images

In an image element, the TCS's can change only the contexts of the included image entity. In other words, they perform no modifications to other web components. For example,

<img src= "*graphics_file***" TMOD="tomorrow">**

If the page is invoked using a hyperlink that has the context modifier **yesterday** (ie. *the client's request time –1d)* and the image link has the local modifier **tomorrow**, the image corresponding to the client's request time will be instantiated. *(ie. client's request time –1day+1day)*

Temporal Context Switches in Server Side Includes (SSI)

As mentioned in Chapter 2, SSI is a feature that the APACHE server supports. It allows the author to include an HTML file in another HTML file, instead of copying the same contents. It facilitates web content creation and helps with version control by avoiding redundant code scattered among separate files. Temporal SSI is similar but also allows multiple variants in the time dimension. The TCS is inserted into a SSI component and modifies the local context of the entity. It has the following format **<!-- #include virtual = "***included_file***" TMOD = + *offset*>**

The *offset* can be specified using the *six* time units: *year, month, date, hour, minute and second* and is relative to the default temporal context of the *TCS*. (See 3.2.1 for The BNF format.)

A local temporal modifier and an absolute time point constitute an extensional representation of a SSI with TCS. Given **<!--#include virtual="file.html" TMOD=" $July 5, 98 12:00:00+7d +4s" >** , an instance of "**file.html**" at the

time context: *July 12, 98 12:00:04* should be included in the HTML page generated.

4.2.3.2 Web Intensions and Web Extensions

A web intension is the relative expression of a web context in terms of a default context. It is specified in a temporal web element with a temporal modifier; while *a web extension* is an absolute temporal context value. It is represented using an absolute time and an optional temporal modifier.

As introduced in Chapter 2, the same intensional expression with the same literal expression can infer different extensional values depending on the context. Consequently, the same temporal URL requests can be resolved into different instances (conventional HTML pages) depending on the time context.

4.2.3.2.1 "Now" Discussion--Intension and Extension

Strictly speaking, there are 24 extensional representations for *now* because of the 24 time zones in the world. For example, given the following TST:
`<a href="somewhere" TMOD= "+1d-2h" >`, and the client lives in a place where "now" is *July 13, 1998, 3:30:00* and he/she issues a THTML request to a server in another time zone, whose time is "*July 13, 1998, 6:30:00*", which time should be chosen as the default temporal context?

In order to deduce the right answer, let us examine the request process at the server. The user-request time is sent to the server in GMT (Greenwich Mean Time) format according to the request protocol. Since no mechanism is provided to retrieve the reader's time zone, then the local representation at the server becomes the default temporal context.

Therefore, the time related process at the server side is as follows:

When the server receives a THTML file, the reader's request time is computed and converted into the server's local time (which forms the basis for retrieving the best-fit version). The instantiation process begins after the global context is resolved. In the previous example, the default time context is therefore *July 13, 1998 6:30:00+1d-2h=July 12, 1998 4:30:00* .

If a reader expects to retrieve an HTML page at the server with the same

extensional time representation as the client, the extensional representation of *now* at the client's side must be written into the temporal request. For this case, it should request

```
TMOD="$Fri July 12, 1998 3:30:00 -1d+2h"
```

The process of conveying temporal information between the web client and the server is illustrated with the following graph:

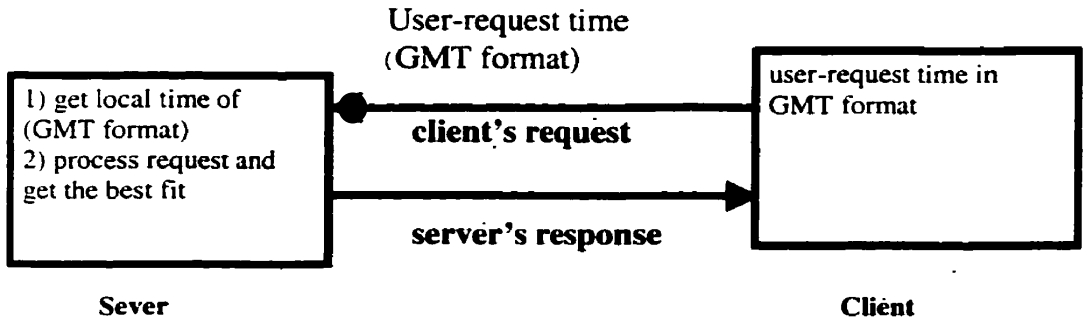


Figure 4.5 Time Conveyance in THTML System

4.2.4 Conclusion

With the incorporation of IL, the server's functionality can be perceived as converting an intension (the URL or THTML tag) into its extensional representation (the web page generated in response). Temporalized web elements are intensions that use TCS's to switch among different possible worlds. Web authors are allowed to define multiple source files for each element. Their TIS's determine if it is adopted as the extensional representation of the element at a particular time context. Due to the above designs in syntax and semantics, the range of HTML pages viewable at the same URL is increased dramatically from the client's perspective. 4.3 Backend System Design

4.3.1 Design Requirements

One of the design requirements was to make the THTML language independent of web browsers. The browser should not know whether or not the server is a THTML or regular HTML server. So the temporal enhancements to the language must be supported only through the backend. In order to support THTML, the server must implement the following:

- The system should have a mechanism for maintaining multiple THTML source

files for each web component.

- It should also maintain the hierarchical relationship between all THTML sources and their respective components.
- A time tag is associated with each THTML file, so that the server can determine which file to retrieve based on the time context.

The above requirements are met in THTML by implementing a *tree* structure to maintain the hierarchical relationship, adopting a naming convention to differentiate between multiple source files and including TIS's in each file. Each of these aspects will be discussed further in the following sections.

4.3.1.1 Tree Structure

First, in file system design, a *tree* structure is used to maintain the hierarchical relationship between THTML source files and their respective web components. A normal URL indicates a directory in the THTML system that holds multiple THTML sources for that URL.

There are other mechanisms to maintain the hierarchical relationship in version space besides the *tree* structure. One commonly used method is to keep the *file* that contains the relationship between the THTML sources and web components. There are several reasons why it was not adopted. First, the web server would have to access the file for each instantiation process. The file would become a key component of the system and would requiring special maintenance. If it became corrupted, the whole system would shutdown. Furthermore, if a search for a target THTML source file was executed at the same time that the author was creating a new THTML file, the file would become a *critical section* and a *synchronization* mechanism would need to be adopted.

The tree structure has significant advantages in comparison:

- By embedding the hierarchy into the structure it eases system maintenance
- The algorithm to retrieve the best-fit THTML source is simplified as well.

4.3.1.2 Naming Convention of THTML files

Second, a simple naming convention needs to be adopted to maintain the parent-child relationship between a temporal object and each THTML file. A straightforward

naming convention is described below.

When a file is created, it is put in its module's directory. For example, all THTML sources for the web component **home.html** are put under directory **home.html**. The format of a THTML source file is *encoded_filename.n*.**html**, where *n* is a number generated automatically by the system when the file is created. It reflects the order of creation and distinguishes the different THTML files. An HTML instance, which is an extensional expression of a web component, has a file name ending with the *suffix* **.html**. It is similar to a conventional HTML file, but contains an encoded temporal context that is inserted between the file name and the suffix. For example, **home@hOdIAYwbySqzGuTi_Rgej_nPkeai.html** is an instance of the module **home.html** at the relative temporal context: **OSOIOH-1DOMOYOW**. The format of an instance is *module_name*@*encoded_context*.**html**. The encoding scheme helps the algorithm retrieve the best-fit THTML file.

4.3.1.3 Design of Time Interval Stamp (the Time Tag)

Third, the TIS is used to relate time to a specific THTML file. It indicates to the system which file is most relevant to a temporal request.

There are many reasons for keeping a time-interval stamp in a THTML file.

- In order to instantiate an HTML page, the system needs to keep time information for each THTML source file. Keeping the information in the file provides a straightforward way to link the time and source page.
- It reduces the time to retrieve the corresponding time interval of a THTML source file.
- It provides a natural way to combine THTML files with regular HTML files. HTML files are just THTML files with time intervals from *negative infinity* to *positive infinity*, which is denoted with time label ⋆ ⋆ ⋆ ⋆ ⋆ ⋆ ⋆.

The TIS is the key to temporal design. It not only specifies the time relevancy of each file, but also reflects the refinement relationship between the different THTML files.

4.3.1.4 Conclusion of File System Design

The result of combining the tree structure, naming convention and time interval stamp is shown in Figure 4.6. Each URL corresponds to a module and therefore has its own directory. Each directory contains all user-defined and temporal source files associated with the module. They are stored using the naming convention *<module_name>.<encoded_userdefined_version>. n*.**html**. Although it is currently implemented on one computer, the design can be easily incorporated into a distributed file system.

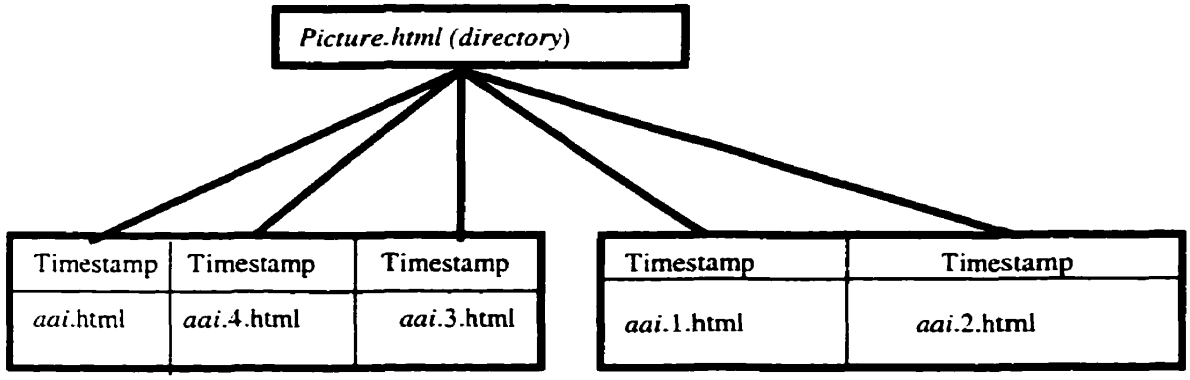


Figure 4.6 System Structure Design

4.4 Algorithm Design

4.4.1 THTML Web Server Model

In a typical client/server model, when the client selects a temporal element on a THTML site, a connection is created between the client and server. The client then issues a request for an HTML file with a particular time point to the server and the server responds with an instance at that time point to the client.

In this model, the web server can be perceived as a black box. With the request of a temporal HTML file as input; the server outputs its extensional representation.

During this process, a time irregularity-handling algorithm is used to obtain the correct local context from both the temporal request and the local modifiers. A temporal

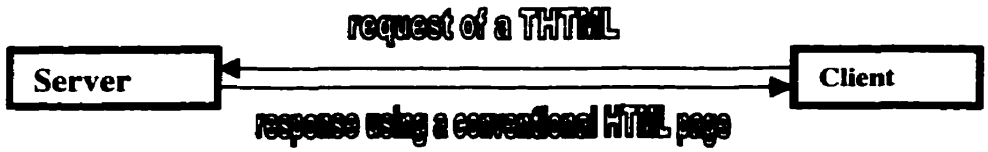


Figure 4.7 THTML Working Model

versioning algorithm is then used to retrieve the best-fit THTML file. These algorithms will be described in the following section along with the incorporation of *push* technology to keep the clients updated with the most current HTML pages.

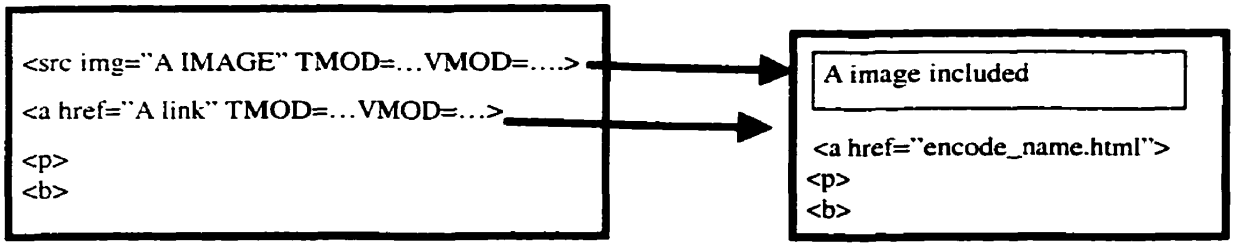


Figure 4.8 Interpretation from THTML to Conventional HTML

4.4.2 Time Irregularity, User-defined Temporal Terminology and Hierarchy of User-Defined Calendars

4.4.2.1 Gregorian Calendar and its Irregularity

Calculations related to time have an irregular feature. For example, **`$ Fri July 31,1998 + 1d`**, should be *August 1, 1998* rather than *July 32, 1998*. The time irregularity is caused by the Gregorian calendar people use as an international standard for civil use.

A calendar is a system used to organize units of time for the purpose of reckoning time over extended periods. By convention, the day is the smallest calendarical unit of time; the measurement of fractions of a day is classified as timekeeping. The generality of this definition is due to the diversity of methods that have been issued in the development of calendars.

In the Gregorian calendar, years are counted from the initial epoch as defined by

Dionysius Exiguus, and are divided into two classes: common years and leap years. A common year is 365 days in length; a leap year is 366 days, with an intercalary day, designated February 29, preceding March 1. Leap years are determined according to the following rule:

Every year that is exactly divisible by 4 is a leap year, except for years that are exactly divisible by 100; these centurial years are leap years only if they are exactly divisible by 400.

As a result, the year 2000 is a leap year, whereas 1900 and 2100 are not leap years. These rules can be applied to times prior to the Gregorian reform to create a proleptic Gregorian calendar. In this case, year 0 (1 B.C.) is considered to be exactly divisible by 4, 100, and 400; hence it is a leap year.

The Gregorian calendar is thus based on a cycle of 400 years, which comprises 146097 days. Since the number 146097 is evenly divisible by 7, the Gregorian civil calendar exactly repeats after 400 years. Dividing 146097 by 400 yields an average length of 365.2425 days per calendar year, which is a close approximation to the length of the tropical year. Comparison with Equation 1.1-1 reveals that the Gregorian calendar accumulates an error of one day in about 2500 years. Although various adjustments to the leap-year system have been proposed, none has been instituted.

Within each year, dates are specified according to the count of days from the beginning of the month. The order of months and number of days per month were adopted from the Julian calendar as follows:

January 31 February 28 March 31 April 30 May 31 June 30

July 31 August 31 September 30 October 31 November 30 December 31

In a *leap* year, February has 29 days.

Based on the representation of time keeping and the Gregorian Calendar currently used internationally, the default hierarchical concept of the time units used in THTML is defined as $second \leq minute \leq hour \leq day \leq month \leq year$. A set of functions supporting the above concept is provided to guarantee the correctness of time calculations, which are discussed in next chapter.

4.4.2.2 User-defined Temporal Terminology and Hierarchy of User-Defined Calendars (not implemented yet)

- **User-Defined Temporal Terminology**

As the design in further implementation, the TCS can be enriched with more complex temporal language with enhancement of additional backend supports.

For example, additional religion dates say *Easter* can be recognized in a TCS by incorporating the following operations to calculate the date of Easter in Gregorian calendar in THTML.

Y stands for Gregorian year. All variables are integers and the remainders of all divisions are dropped. The final date is given by two values: M (the month) and D (the day of the month).

$$C = Y/100,$$
$$N = Y - 19*(Y/19),$$
$$K = (C - 17)/25,$$
$$I = C - C/4 - (C - K)/3 + 19*N + 15,$$
$$I = I - 30*(I/30),$$
$$I = I - (I/28)*(1 - (I/28)*(29/(I + 1))*((21 - N)/11)),$$
$$J = Y + Y/4 + I + 2 - C + C/4,$$
$$J = J - 7*(J/7),$$
$$L = I - J,$$
$$M = 3 + (L + 40)/44,$$
$$D = L + 28 - 31*(M/4).$$

By the same token, the user-defined calendars and the temporal concepts can be specified with corresponding backend support. For example, if a student can use the term *semester* and *academic year* in his TCS, operations of parsing these terms and converting them into Gregorian calendars should be provided.

- **User-defined Calendars and their Hierarchy**

Further, if multiple user-defined calendars are specified in the system, ordering should also be specified among them to eliminate ambiguity. For example, *New Year's Day* has different Gregorian values in the Lunar Chinese Calendar and the Jewish Calendar. If a temporal request having a term *New Year's Day* is received, ambiguity must be resolved if both calendars are specified in the system. Therefore, an order should be specified between them. In the future, an intensional tier for user-defined calendars, their ordering and operations should be provided by the system.

The relationship of components in the intensional tier is as follows:

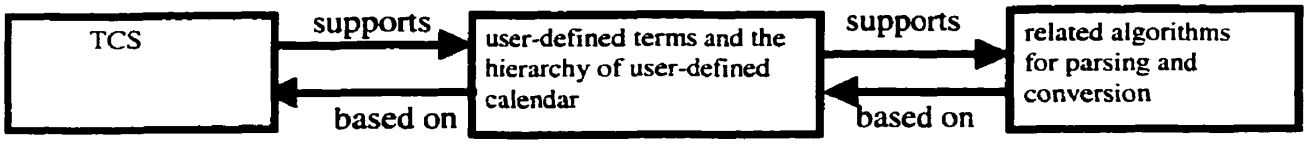


Figure 4.9 User-defined Calendars, Orders and Operations

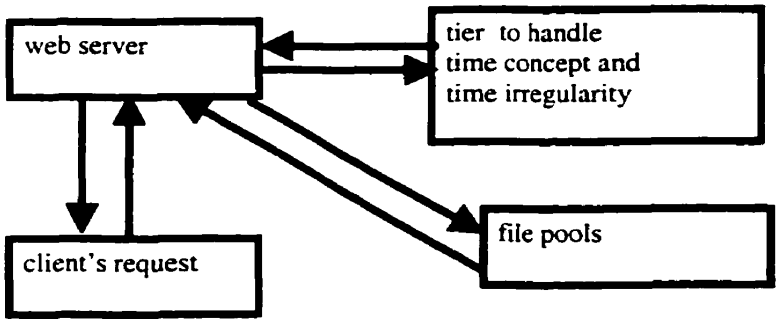


Figure 4.10 Intensional Tier Supporting User-defined Time Concepts

The relationship of the intensional tier with other components in the system is illustrated in Figure 4.10. The server will retrieve the temporal concepts from the intensional tier and compute the results.

4.4.3 Temporal Versioning Algorithm

The purpose of instantiation is to generate an HTML instance in response to the URL request. After the time context is extracted, the server will search for all THTML

source files that match the time point specified. If there is only one match, that file will be used to generate the HTML instance. Otherwise a best-fit file is chosen using the intensional and extensional rules described in Chapter 3. If the rules cannot resolve all the ambiguities, then the absolute (default) THTML source file is used. This file is just an ordinary HTML file with no time stamp. It is kept in the system as a last resort when no time intervals match a specific time point.

The THMTL source file is then parsed. Any links, images and other tags with temporal attributes are also instantiated recursively until all temporal elements are converted into their respective extensional expressions.

4.4.4 Push Technology and Dataflow Model of WWW

With the rapid growth of time sensitive information on the web, the reader needs to be periodically updated with the most up to date information. This is facilitated in THTML with the incorporation of *push* technology.

4.4.4.1 Problem of Cache in the Existing Web Model

The cache is one of the most significant features of the major browsers (Internet Explorer and Netscape Communicator). With a cache, copies of responses are stored in the local file system and can be reused for the same URLs without successive requests to the server. Therefore, caching conserves bandwidth and reduces network latency. However, if the caching mechanism not implemented appropriately, there is a risk of receiving stale data from the cache.

In order to enhance cache control, the HTTP protocol contains both explicit specifications as well as heuristic hints in HTTP/1.1, concerning the cacheability of documents and how often their freshness should be verified in order to guarantee that they are still up-to-date.

One of the most important features in HTTP/1.0 about caching is the conditional GET. It allows a document to be retrieved conditionally, based on whether it has been modified since the last access. If the document has not been changed, a very short **`not modified`** message is issued; otherwise, the updated document is transferred. For

example, after a client requests a document **/some/where/foo.html**, the following response can be sent to the client:

```
HTTP/1.0 200 OK
Server: Netscape-Enterprise/2.0
Date: Sat, 19 Apr 1997 10:22:00 GMT
Last Modified: Fri, 18Apr 1997 15:12:05 GMT
Content-type: text/html
Content-length: 6510
```

The above headers indicate the server and the version of HTTP protocols. The time the web transaction happens, the last modified date of the file. In the subsequent requests, a conditional GET request now adopts the timestamp from the **Last-modified** header and issues it along in the request header with an **If Modified Since** header to the server. The example of the subsequent requests is as follows:

```
Get  /some/where/foo.html HTTP/1.0
If Modified Since: Fri, 18Apr 1997 15:12:05 GMT
Accept: text/plain, text/html
```

It indicates that if the document is not modified since the date defined in the **If Modified Since** header at the server side, the document content is not issued to the client. Therefore, the following headers but contents will be sent the client as the response.

```
HTTP/1.0 304 Not Modified
Server: Netscape-Enterprise/2.0
 Date: Sun, 20 Apr 1997 15:45:12 GMT
```

However, if the document has changed, the server will feed the client with an updated document and a **200 OK response. 304 Not Modified** response saves bandwidth and reduces latency, as no document transfer actually occurs.

The above mechanism is efficient in conventional HTML to keep the most updated content in a client's browser; however, in THTML file system, as the time changes, it is

usually a separate file (i.e. a different version) that should be issued to a client because the original document is not changed and stored as a historical version in the system. Therefore, the header indicating whether the file has been modified cannot prevent clients from viewing obsolete versions in THTML system.

4.4.4.2 Possible Solutions

Based on the above problems, two solutions are provided. The advantages and disadvantages of them are discussed in the following sections.

- **Cache Control**

The first approach is to delete the cache. **`HTTP/1.1`** allows a client to specify when the file expires and whether he/she uses cache or not. A typical adoption is web commercials. Normally, THTML documents do not change while the images within inline ads are. Therefore, the HTML content is not modified on a per-request or per-user basis in order to contain a different ad. Instead, only the inline images update with a particular frequency. With this approach HTML pages can be cached. Only the images themselves force successive requests to the server instead of being served from the cache. Although the contents can be updated with this mechanism, the disadvantage of this approach is that substantial web traffic is induced. Therefore, it is not feasible for it to adopt in a large, *dynamic* HTML file applications.

- **Server Push**

The other approach is to complement existing web protocols with a new technology----*Push* technology (compared to *Pull* technology), which is considered a more promising solution.

The original *pull* technology (*client pull*) refers to a model in which a client requests, then the server responds while *push* (server push) refers to a system in which a client automatically receives information or applications from a network server. With *server push*, clients do not have to issue successive requests to the server for the most updated versions.

In a *server push*, the connection between the server and a client does not end until

the client closes the session. The server periodically feeds the client with the most updated version. Because multiple connections need to be kept and the responses to the previous connected clients should be issued periodically, server push burdens the server. However, the approach is preferred in THTML. The advantage of it is that web traffic is reduced to almost half (theoretically for lack of successive requests from the client). It was shown in a year-long study involving 4,000 users in six Fortune 1000 companies that just less than 20% of network traffic was raised with the use of push technologies while 100% with the use of the client pull technology. Another significant reason that *server push* is preferred is that no particular software is required with the client, because the functionality of THTML is enhanced into the THTML web server.

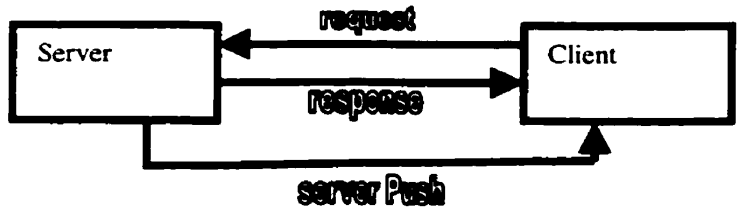


Figure 4.11 Server Push

With this technology, synchronization can be achieved between a client and a server with a pre-defined frequency. It not only avoids the likelihood that a client may be viewing the stale information but also reduces web traffic tremendously. Although today *push technology* is still in its infancy, many commercial push products have been available within industry, and web technology is taken into a promising new research direction. With *time* addressed, an ideal solution is provided with push in order to bring the increasingly *dynamic* feature to the World Wide Web.

4.4.5. Conclusion of Algorithm Design

For convenience, THTML allows the author to specify time using the Gregorian calendar. This however creates a problem when adding or subtracting time due to the irregular way in which time is denominated in this system. We have therefore implemented a time irregularity-handling algorithm to perform these operations correctly. This algorithm can be extended in the future to handle different calendars and

temporal terms.

Once the time point has been determined, the best-fit time interval must be found. The temporal versioning algorithm uses both the intensional and extensional rules to determine this best-fit interval and resolve any ambiguities.

Push technology was also incorporated into THTML to facilitate the continual update of a reader's page. It involves executing the above calculations at user-defined intervals. Although this results in a heavier load on the server and other solutions may exist, it is the most efficient solution for some applications where up to date information is essential.

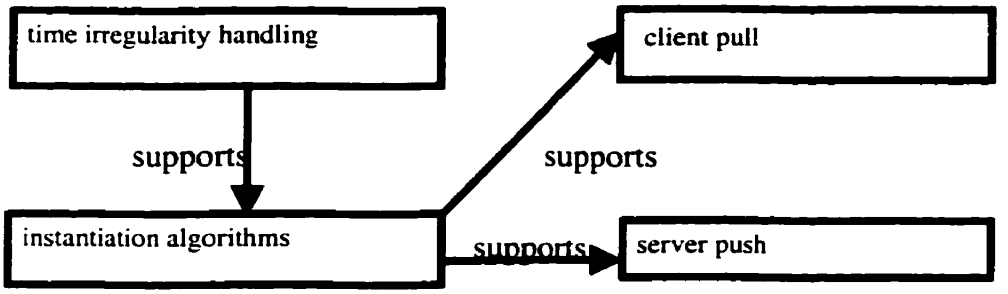


Figure 4.12 Backend Components of THTML

4.5 Combination with IHTML

As a side, THTML1.0 is compatible with IHTML. THTML files of a user-defined version can be created and modified. This indicates a user-defined version can be varied with time in semantics. Clients can request a particular instance of user-defined version of a web component at a particular time point. In a THTML file having user-defined version, the TST's have the following formats:

TST: : = **[<!--#include virtual="***included_file***" TMOD="***TCS***"**
VMOD="*intensional_context_switches***"> <src img="***imgfile***"**
TMOD = "*TCS***" VMOD="***intensional_ context_switches***"> |**
<a href= "*URL_Address***" TMOD="***TCS***"**
VMOD="*intensional_context_switches***" >]***

• The format of *intensional_context_switch* is detailed in [10]. The *TCS* has the same format as introduced in Section 4.2.2.1. Therefore, for example, in a hyperlink element, a client can write,

<a href="*URL_Address***" VMOD="pic:big, bgc:blue" TMOD="yesterday">.**

In THTML1.0, temporal instances are implemented as special *subversions* of a user-defined version, so when an element having both **TMOD** and **VMOD** attributes is encountered, the user-defined version is first resolved, and then the process of instantiation, namely, the instance of the user-defined version is resolved in the time dimension. The resulting instance is sent to the client*. In the above example, attribute **VMOD** specifies the local, user-defined modifiers: **pic:big** and **bgc:blue** of the global user-defined version of the page. If the global, user-defined context of the page is **lang:english**, the local user-defined context of the element will be **pic:big + bgc:blue + lang:english**. After the best-fit user-defined version at the above context is resolved, the instance of it will be resolved. Attribute **TMOD** modifies the temporal context into *current time-1d* for the instance. The THTML file whose time interval stamp best fits the temporal context is instantiated. It replaces the intensional element into its extensional expression.

In the THTML1.0, multiple HTML files of a user-defined version can be created and modified in the system. The command for creating a new THTML file from a user-defined version has the following format:

ivi [-v *version* **[-b** *baseversion***]] [-t]** *module*

It indicates that the author can specify a THTML file of a user-defined version of the module `module`. The option **[-v** *version* **[-b** *baseversion***]]** is used in the same manner as in IHTML 2.0 to define a user-defined version. If the `version` version does not exist and the base version is supplied, it copies the *baseversion* version of the page to the `version` *version*.(see the manual of IHTML 2 [10].) A THTML file of the user-defined version will be created if there is option **-t**; otherwise a

---

• The semantic of versioning resolution principle is omitted, because the version resolution principle will incur more ambiguities.

new user-defined version is created. For example, by the command: **ivi -i bgc:blue -t home.html** a unique THTML file of a user-defined version **bgc:blue** of a web component **home.html** is created.

In the existing file system, both the user-defined version and its THTML files are put in the directory of their parent module. The name convention of a THTML file of a user-defined version is *module_name.encoded_user_defined_version.n*.**html**. The user-defined version information is encoded in encoded_user_defined_version. Variable *n* indicates the temporal version of the user-defined version. For example, **home.M11DmCqOxicq.1.html** indicates the THTML source file for *english* version of **home.html**.

4.6 Conclusion and Further Directions of THTML System Design

One of the guiding principles behind the design of this THTML system was to leverage the existing web architecture and HTML language. The language was extended by adding TST's and TIS's. These two enhancements allow web authors to specify which pages or components should be instantiated given a URL request from the reader. These times can be conveniently specified using relative terms like, today and tomorrow or in absolute terms using the Gregorian calendar. Internally, these time elements are converted to six time units and comparisons between the time request and time intervals are performed using these units. Both the intensional and extensional rules are used to resolve any ambiguities and the results of a match are sent to the client.

Since many browsers already exist that support conventional HTML, the system was designed to be independent of any HTML browsers. All the functionality for determining which page should be instantiated in response to a request is located at the server. The server stores the THTML files, along with its time interval stamp, in a tree structure with a naming convention that encapsulates the parent-child relationship between the modules and the different temporal versions. Although it is less efficient than storing all time stamps in a single file, it is more robust and facilitates implementation on a distributed file system without any bottlenecks.

The server is also responsible for interpreting requests given by the user and

generating an HTML page in response. Since users specify time using the Gregorian Calendar, a time irregularity-handling algorithm needed to be implemented to handle the

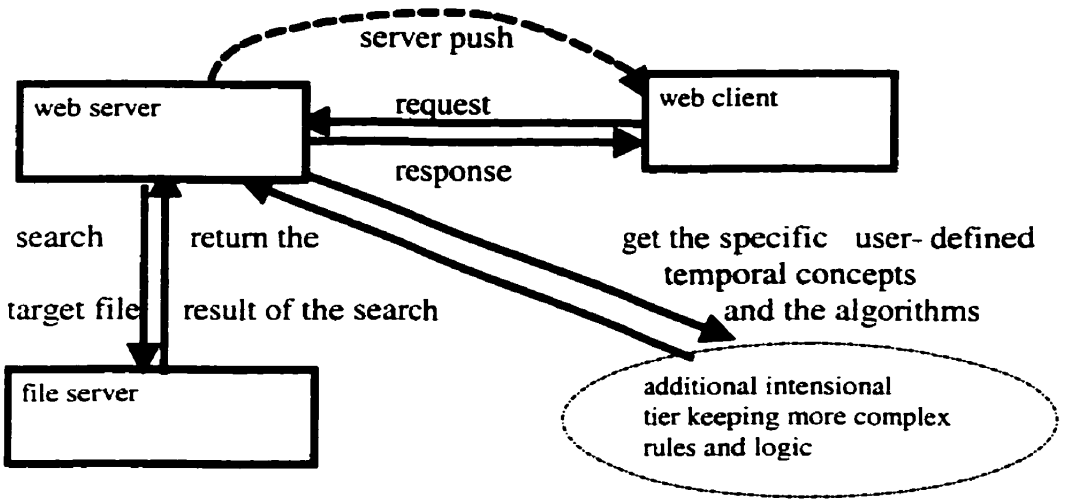


Figure 4.13 Three-tier Model of THTML System

irregularities when performing addition and subtraction operations. The server uses this algorithm to determine the correct local context from the inherited temporal context and a tag's context modifier. The temporal versioning algorithm uses this local time point to find the best-fit time interval. The generated page is sent in response to the client's request. An updated page can also be sent at user-defined intervals using *push* technology. This technology leaves the responsibility with the server to keep the client up to date with the latest information. The three different tiers are shown in the graph above along with the communication among them. Additional tiers can also be added to increase the functionality of the system. Further enhancements to the system can include the following aspects:

- At the front end, more expressive TST's can be added, including more user-defined temporal terminology.
- At the backend, the file system can be separated from the web server. Due to the exponential growth of versioned entities stored in the file server, the THTML file system should be made distributed.
- On the web server, another improvement would be to incorporate an additional

intensional tier that includes more complex temporal rules that can be used to satisfy additional requirements from web authors.

- The server push technology can be further improved by including a friendlier user interface for web authors and more advanced protocols at backend to reduce server load.

Chapter 5

THTML Implementation

5.1 Overview

In THTML all functionality is incorporated into the web server. The web client issues conventional URL and receives conventional HTML file from THTML server. THTML is implemented independent of web browsers.

We will introduce the implementation of the functionality covered in Chapter 4. This includes the parsing of THTML files, the temporal versioning algorithm, time irregularity handling and the generation of conventional HTML pages, along with the data structures and the *push* implementation. They are discussed in the following order: the THTML request handler and response handler will be discussed first. Then the data structures passing between both functions are discussed. This is to keep user-defined and temporal versions. The implementation of *push* technology is then discussed. Finally, the chapter concludes with an example site written in THTML.

5.2 Design Requirements

THTML1.0 implements partially the designs presented in Chapter 4. With THTML1.0, TIS's can be specified in an HTML file, which are the time the author intends to post the file at. The TST's can also be defined, which can be either intensional or extensional (including absolute time points) expressions. *Server push* and *the customized calendars* are not implemented at the current stage.

In THTML 1.0, after the reader issues a request to the web server, a particular instance, namely conventional HTML will be responded to the client at a given time. The processes at the server are expanded as follows:

- **Parsing the global context**

First, the temporal context should be extracted from the URL. This context is used

by each component of the THTML page as the default global context.

- **Time Irregularity Handling**

Time related operations are needed to retrieve the correct time context for a request. These operations are as follows:

First, the GMT format of the user-request is converted into a local time before the server can compute the global context.

Next, the local modifier and the global context are combined to retrieve the local context for each temporal element.

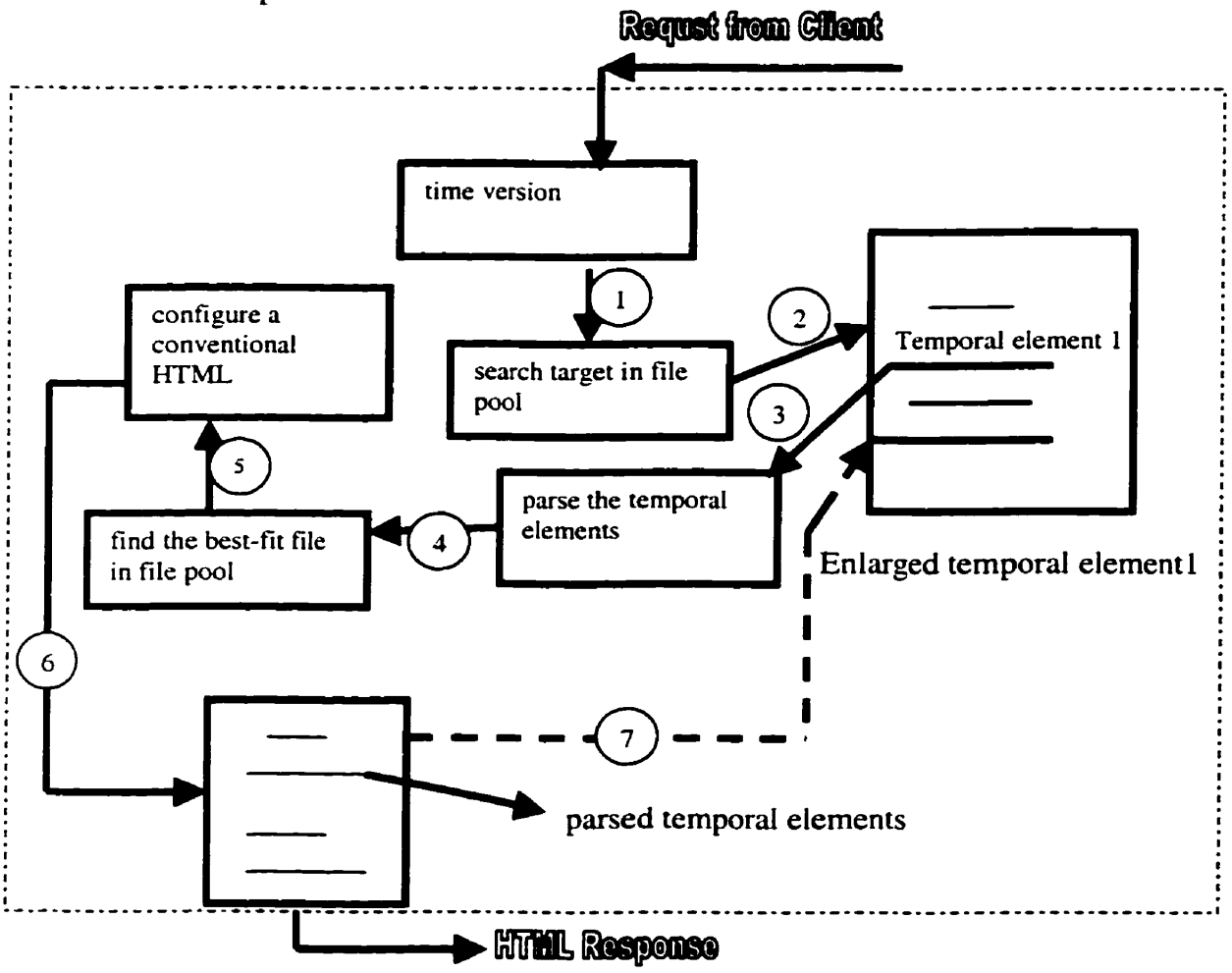


Figure 5.1 Enlarged THTML Web Server

Because of the irregularity of the Gregorian calendar, special *time irregularity* algorithms are used to obtain the correct temporal context.

• **Instantiation**

The process is illustrated in Figure 5.1. When a temporal component is encountered in a THTML file, it is converted to an *extensional* representation.

First, the global context and its local context modifiers are combined according to the rules introduced in Section 4.2.3.1.1. After the local context is obtained, if the component is an image or a SSI, the server will search for the corresponding THTML source and generate its instance. This instance will then be inserted into the newly created HTML page. The process of instantiating an instance of a temporal component is the same as converting a web component with TCS into its extensional expression. Steps 3 to 6, above, repeats until all the components with TCS are converted. When instantiating a component, the process above is applied recursively.

5.3 THTML Implementation

5.3.1 Hook-up with APACHE API

As was introduced in Chapter 2, THTML is implemented by extending a module on the *Apache Server API.*

The extended module structure is as follows:

```
module thtml_module = {
      STANDARD_MODULE_STUFF,
      thtml_init,         /* initializer */
      NULL,  /* per-directory config creator */
      NULL,/* dir config merger-default is to override*/
      thtml_config,      /* server config creator */
      NULL,              /* server config merger */
      thtml_commands,    /* command table */
      thtml_handlers,    /* [8] list of handlers */
      thtml_xlate, /* [1] URI-to-filename translation */
```

```
    NULL,      /* [4] check/validate HTTP user_id */
    NULL,    /* [5] check HTTP user_id is valid *here**/
    NULL,   /* [3] check access by host address, etc.*/
    NULL,       /* [6] MIME type checker/setter */
    NULL,     /* [7] fixups */
    NULL,       /* phase [9] logger */
    NULL        /* phase [2] header parser */
};
```

The above structure module is the link between functions defined by third party and the APACHE API. It defines the 9 phases that the server goes through after receiving a URL request. In each phase, the third party can specify a handler to customize the server's behavior at that phase. NULL indicates that the third party is not interested in handling that request in that phase. Consequently, from the above THTML structure module, one can see that there are totally two handlers specified to handle THTML requests, namely **thtml_handler** and **thtml_xlate,** at step [1] and [8] respectively.

Each handler is a function taking as an argument, a pointer to a structure called **request_rec**. This structure contains all the information for a particular request, including the **user_request_time** in GMT format and whether the document sent to the client should be cached or not.

It also contains information to be sent to the next phase. The **thtml_xlate** handler adds user defined version information to the structure to be passed to the **thtml_handler**.

In each phase, each handler, when invoked to handle a particular **request_rec**, has to return an **int** to indicate what happened.

OK --- the request was handled successfully. This may or may not terminate the phase.

DECLINED --- no erroneous condition exists, but the module declines to handle the phase; the server tries to retrieve another.

An HTTP error code--- it aborts handling of the request.

Therefore, the request and response handlers in THTML 1.0 have the same data structure **request_rec** as input parameter and both handlers return a status code. The following sections give the functionality of them.

5.3.2 THTML Request Handler--thtml_xlate

The request handler is the first phase of the server. The function is to handle the user's request by translating the URL to a filename.

First the URI is translated into a filename, the server will search if it is a THTML request by parsing out the version information in the filename.

If the file is a versioned entity, the server first checks whether it embeds user-defined context and temporal context. If yes, it will then parse the user-defined version information first and then the temporal context information. An error message will be returned if it cannot be parsed.

The next step involves applying the best-fit algorithm to find the best-fit THTML file. The following steps are involved in this step.

The URI is translated into filename first, the filename is judged. If it is a THTML directory or a THTML file (implemented in **i_is_userdir**), the temporal contexts and the user-defined version information if any are parsed out. The request time point is parsed out if there are no time contexts in the filename.(in **translate_userdir**) With the time point and the time contexts in the filename, the server finds the best fit file (in **get_time_vers**). The above steps are illustrated with Figure 5.2. In the **get_time_vers** function, the following processes are adopted to find the best-fit file: **time_fit** function is adopted first to judge if each time stamp in a THTML file matches the time point in the request. If yes, put the result pattern into a two dimensional array. The all the time patterns in a file are checked, **find_least_gap_time_pattern** is adopted to determine the best-fit among all the candidates.

During the above process, if the best-fit file cannot be found, the server will write an error message to the log file and return the error code **HTTP_NOT_FOUND**.

Otherwise in step 3, the temporal version will be resolved in two steps. First the time context is obtained from the THTML input, second we search for the best-fit temporal version.

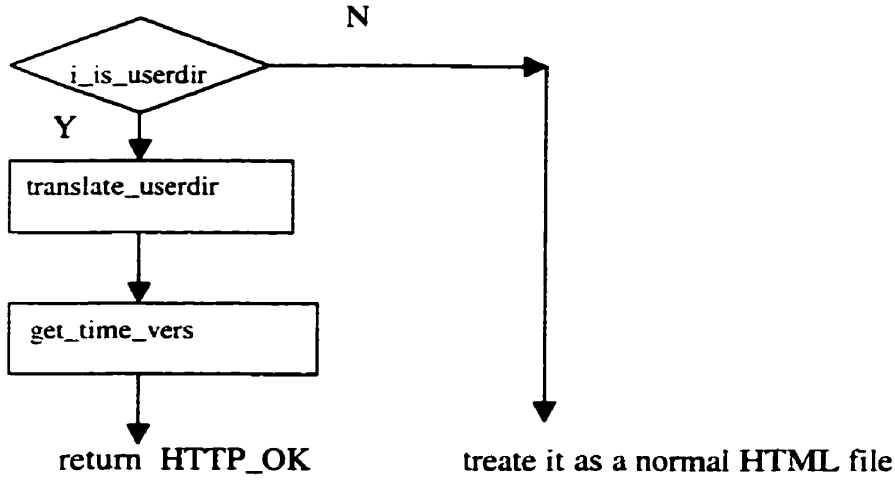


Figure 5.2 Flow chart in the Request Handler

If the target THTML source file is not found and the absolute THTML file does not exist, the error message **HTTP_NOT_FOUND** is given. The error is also written to a log file corresponding to the THTML file.

Before an **OK** is sent, the user-defined and temporal context information is put into the data member **user_request->config**. The response handler can use it as the global context for interpreting local temporal elements in the file.

5.3.3 THTML Response Handler-- thtml_handler

thtml_handler is the response handler, which handles the **MIME** type "**text/html**". It is the last function to be executed before a response is sent to the client.

Its functionality is to translate the intensional elements in the best-fit file resolved in the request handler into extensional representations and send the resultant conventional HTML page to the client. It only handles versioned entities *SSI*, *image* and *hyperlink*. **DECLINE** is sent if the **config** field in the user request is NULL. This indicates that no version information was given.

First the server opens the best-fit source file that was found by the request handler

and begins to parse it. It is stored in the data structure **requst_rec->filename**. When it meets a temporal or user-defined element in the file, it is parsed into an extensional expression. In the first step, if the local context is not defined, then the global context will be used instead. It is stored in the **request_rec** data structure. It performs the same procedure recursively on any included files. First, it checks if it is an IHTML file. If yes, it will parse the version information and extract the relevant parts of the file. The function **get_time_vers** will be adopted again to retrieve the best-fit file based on a given temporal context. The above steps are illustrated by the following figure:

```
While(ts=I_sub_translate)
{
    case ISELECT: do_select;
    case ICOLLECT: do_collect;
    default;
}
```

in the **i_sub_tanslate** function, if a image, hyperlink and SSI are met, their temporal contexts will be parsed out, and then the best-fit will be found. If other tags are encountered, they are interpreted the same as the manner in conventional HTML.

```
While(next_tag) {
 if(interesting_tag)
{
  Fix_tag,
  Print_tag
}
else if(cron tag )
       Ignore the content in it and continue parsing
else if(include_tag)
       Do_include
else
```

```
        if(!skep_content)
              Print_tag
}
```

The interesting tags are the three tags that may include the temporal context switches. In the **fix_tag** function, if a **VMOD** or **TMOD** attributes are met, their values are extracted and merged with the global intensional and temporal contexts. The resulting context is encoded into the element name. For a *SSI* and *image* tag, the best-fit file will be retrieved and included.

5.3.4. An Example

When readers go to `http://valdes.uvic.ca:8080/~lliu/homepg/`**home.html**, the following steps are performed in the **thtml_xlate** phase. The directory `/~lliu/.www/homepg/home.html` is checked to determine if the directory allows global read/execute access. Then it checks if it is a directory holding multiple versions. By checking if it has a suffix **.html** and it is a directory, then goes to next step, check if it contains any version information. Because **home.html** carries no temporal contexts. The request time is retrieved as global context. Files in this directory are searched for best-fit to the request time. In this example, the file aai.1.html is the best-fit. The file name along with the user-defined contexts and temporal context are stored. Then the **thtml_handler** will be executed. First the best-fit file will be opened, whose name is obtained in **thtml_xlate,** namely **aai.html** in this case. In this file, when the single temporal hyperlink **<A HREF="home.html" TMOD="yesterday" VMOD="lang:english"> yesterday's english version </A>** is met first, the value of the local modifier in **VMOD** is "**lang:english**" while there is no global version. Therefore, the archor's version will be changed to **lang:french**. The archor's name will be encoded as *encoded<lang:english>@encoded<-1D>*.**html** When the user clicks the above link, the local context will be the global context of the new page. The same steps are applied to parsing the new page.

5.4 Data Structure

The only data structure passed among the handlers is the **request_rec** *structure*. This structure mainly contains protocol-related information from the client, including cache control. However, there are two fields that are especially relevant to THTML, the **request_config** and the **request_time** fields. The **request_time** is used to store the client's request time in GMT format. The **request_config** field is used to hold the structure **versionp**. This structure is where the *temporal* and *user-defined* version values are stored. The time versions are stored in the **tval** field while the user-defined versions are stored in the **vals** field. The data structure of *versionp* is illustrated as follows:

```
typedef struct vstruct {
        int ndims;
        char tval[50];
        char **dims;
        char **vals;
} *versionp;
```

The **dim** and **vals** variables are used to store the dimension and its corresponding version information. **ndim** indicates the number of dimensions, and **tval** is used to store the encoded time information. For example, if the following user-defined version: *bgc:blue+lang:english+img:big+platform:unix* and temporal context:

dim	val
bgc	blue
lang	english
img	big
platform	unix

Table 5.1 Storage of User-defined Version

1998 Dec 12/11:11:13 are encountered, they will be stored in *versionp* While *tval*

stores

Y1998M12D12H11I11S13 (The format and meaning will be discussed shortly). **ndim** is 4 indicating that there are four user-defined dimensions.

This structure holds the global time and user-defined context, which will be used in the response handler to interpret the local intensional and temporal elements.

5.5 Retrieving the best-fit THTML file

5.5.1 Retrieving Candidates

Function **get_time_vers** is used to find the best-fit interval candidate. The process of finding candidates is the process of finding if the time point in a query lies in the time interval stamps defined in the THTML files.

In order to compare them, THTML puts the parsed time pattern found between the **<cron>** and **</cron>** tags into the following structure.

```
typedef struct tim{
                char* sec;
                char* min;
                char* hour;
                char* dom;
                char* month;
                char* year;
                char* dow;
                int flag;
  }timeS;
```

The fields hold respectively: second, minute, hour, day of month, month, year and day of week. The last flag indicates the time pattern, which can be one of the constants **BEFORE, AFTER, DURING, RECURRING**. THTML encodes the two end points of a time unit into one field and decodes it later when comparing it with a time point. Since

only 2 digits are needed to represent each time unit, except for the year, then we can encode the upper endpoint for each unit as the 3rd and 4th and the lower endpoint in the 1st and 2nd position. The year is encoded in a similar manner except using 4 digits each for the upper and lower endpoint. When comparing times, we always start with the most significant unit, so we compare the *year* first, followed by *month*, *date*, *hour*, *minute* and *second*. Unnecessary comparisons are avoided by discarding elements as soon as one time unit does not match.

There are several different situations that need to be handled by the server. These situations will be described below.

• **BEFORE/AFTER** pattern comparison:

For the BEFORE pattern, when we compare each time unit, the function **get_time_vers** returns a 1 if the time request is less than the time stamp, a –1 if it is greater than the time stamp or continue with the next unit if they are equal. If the 2 time points are exactly equal, we also return an error, namely –1. A similar situation occurs for **AFTER** patterns.

• If a **DURING** pattern is encountered, the corresponding unit will be decoded first.

When the first data type is encountered in one time stamp that cannot hold the data in timestamp exclusively, the matching will stop and the function **get_time_vers** returns –1 to the calling function indicating failure; otherwise, if there is any time unit that contains the corresponding data unit in time point exclusively, the function returns immediately with the value 1. Because no matter what value is in the remaining smaller data units, it is always fits in the time pattern; If either of them overlaps with it, the process of comparison will continue recursively with the remaining time units until it is clear whether the time point fits in the time pattern or not. At the end, it turns out that the tying can not be solved between the two time points, i.e. the time point is overlapped with one of the end point of the timestamp, 1 is returned.

• If a recurring pattern is encountered, the server compares each individual time unit. If a time unit containing intervals is encountered, it will be treated in the same manner as a 'during' pattern. The one thing that distinguishes this pattern from all others

is that if a match is found in a time unit, only the matched value is stored and the other

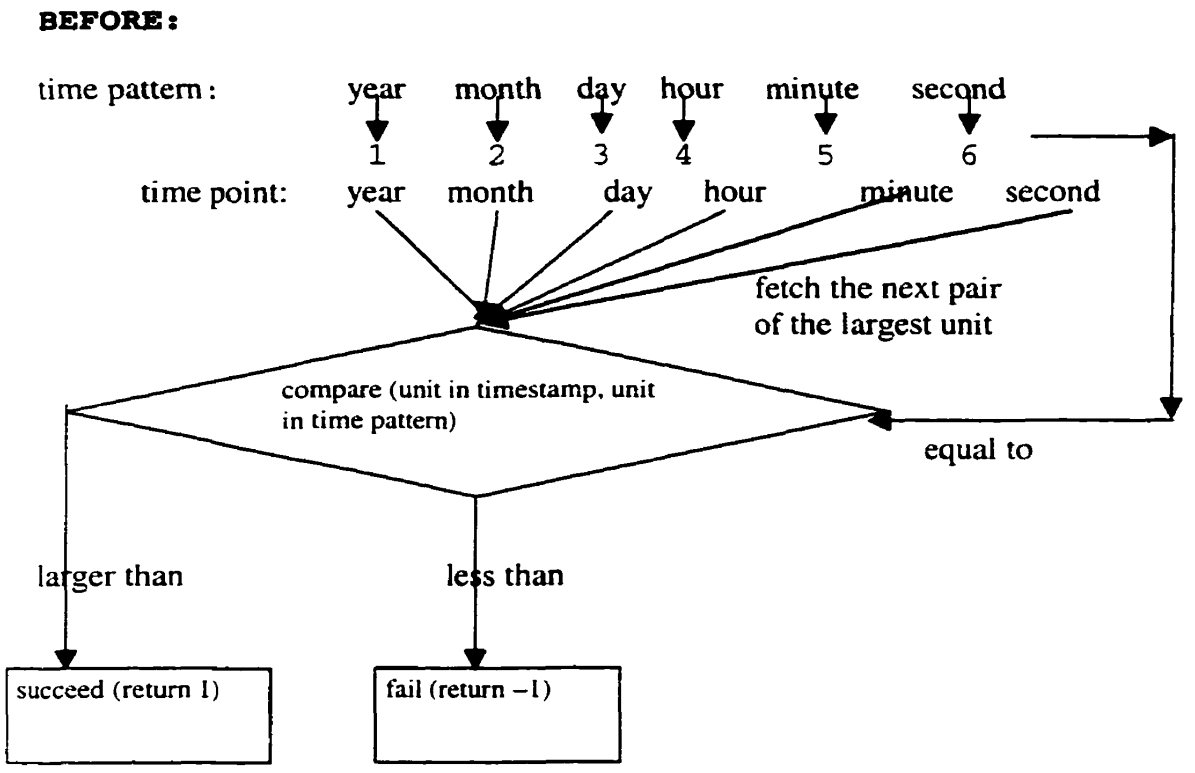


Figure 5.3 Algorithm of Refinement Relationship Resolution between "before" Patterns

recurring values are discarded.

- The situation is simplest in a comparison between two time points. In this case, we just need to match all time units.

If a match is found, the candidate is stored in an array so that it can be compared later with other candidates to determine the best-fit.

5.5.2 Resolving Ambiguity

In the following section, only ambiguities among temporal versions are discussed. The ambiguity of user-defined versions is detailed in the corresponding literature [10].

The ambiguity resolution is handled in the function **`find_least_gap_time_pattern`**, which **`get_time_vers`** uses to find the

best-fit THTML file or the best-fit time interval in a single file. In this function, if more than one pattern matches the time request, the rules introduced in Chapter 3 are applied to solve the ambiguity. The matched patterns are stored in a two-dimensional array. The

year	month	date	hour	minute	sec	pattern
1998	3	4	5	6	7	RECURRING
1998	3	0504	6	6	6	AFTER
1998	3	4	0605	6	6	DURING

Table 5.2 Temporal Pattern Candidates

structure is illustrated in Table 5.2.

The comparison returns the best-fit pattern for further comparison among the candidates, or –1 to the calling function **`get_time_vers`** indicating there is ambiguity, in which case the server finds the absolute THTML file instead as the best fit. If it does not exist, a log is written in THTML log file indicating the vanilla version does not exist.

The server goes through all the candidates, to find the pattern with the highest refinement level. (The order of refinement, from highest to lowest, is *point, during, after/before, and recurring*.)

If only one pattern is found in the server, it will return that pattern.

Note that if one of the matches is a *point* pattern, it is the only one (since time points are unique). So it becomes our best-fit.

- If there is more than one *during* pattern, the difftime function is called to calculate the time interval. The smallest interval is the best fit.
- If both a *before* and an *after* pattern exist the most refined pattern, the error code indicating ambiguity is returned in this situation because there is no way to compare the time gap of these two patterns.
- If the most refinement patterns are either of *after* or *before*, the value of *difftime* between the end point and the target end point is the criterion to choose the best-fit candidate. The one having smaller value is the best-fit indicating that it encapsulates more closely the time point requested.

• If there is more than one *recurring* pattern candidate, the algorithm to find the best-fit is similar to the interval pattern, the time units are compared from the highest to the lowest.

The ambiguity algorithm among files or in one file is the same, except that in the ambiguity handling among files, a file index is kept along with its best-fit pattern in the data structure. When the ambiguity is solved, the corresponding file index is returned so that in the response handler, the server can open the best-fit file.

5.6 Time Irregularity Handling

As mentioned in Chapter 4, special functions (drivers) are provided to handle time irregularity.

The functions provided are based on the Gregorian Calendar:

1. *getDMonth*, which returns the number of days in a given month and year.

2. *getDow*, which returns the day of week for the given date.

• When THTML merges local and global temporal contexts, THTML merges the two time patterns starting from the smallest time unit to the largest, i.e. from the seconds to the years.

• In THTML, the day of the week information is calculated when necessary and so it stores the date using only the 6 time units mentioned above. The reason for this is as follows:

1. to simplify the algorithm

2. to avoid mismatches in the date and the day of week given. For example, if the user gives the time pattern:

```
* * * 5 4 1998 1
```

and April 5 happens to not be a Monday. THTML assumes that the date is correct in this situation rather than the day of week. There are however situations when we need to convert from the date to a day of the week and vice-versa.

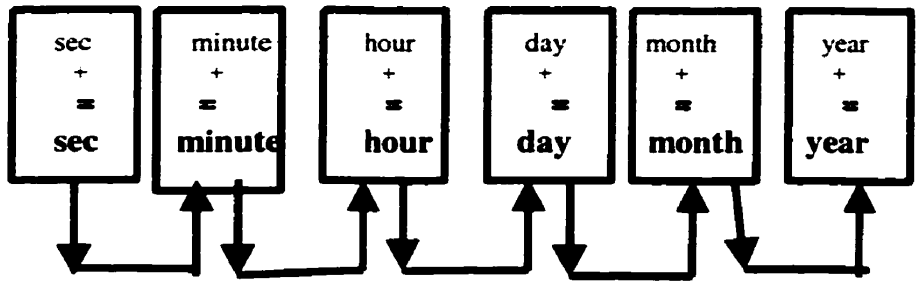
When combining local and global contexts, THTML has to convert the week information given by a user into a date. For example, when a user defines **TMOD="1d+2w"** in his query, it adds *2*14+1* days to the global context.

Another situation is when it meets a timestamp indicating the *dow*, THTML has to calculate the *dow* of the time point in clients' request for comparison.

For example, the timestamp

```
1 2 3 * 5 1998 1,2,3
```

is compared with the time point *May 12,12:12:12,1998*. The THTML server will calculate the day of week of the time point to check if it matches the *dow* given by the timestamp.




```
if overflow:      n/60      n/60      n/24   n/MaxDOfM   n/12
```

Figure 5.4 "Overflow" Among Time Units

5.7 Other Aspects in Implementation

• Version Encoding and Version Decoding

As seen in the design section, THTML encodes the version information in the file name and decodes it when the server retrieves the best-fit version. There are a couple of reasons why it is encoded. The main reason is to prevent the laymen from hacking the system based on the version information. The second reason is to standardize the version information (especially user-defined versions) with standard printable ASCII characters.

In THTML, the *absolute* and *relative* time contexts are encoded using different patterns so that the server can easily differentiate between the two and process them accordingly.

For example, if a link

```
<a href= "A LINK", TMOD="1d-2w">
```

is encountered, the time context is written as **S0M0I0D-13M0Y0** and encoded

following the filename of the link with a separating symbol @ in between.

When the user clicks on it, the server will decode the time information. Since it is relative time, a global context is needed for it to merge with. Otherwise the global time context is ignored and the decoded absolute time is used instead as the time for the query.

- **Implementation of *Server Push***

Server push was not implemented in the current THTML software for the following reason:

In order to implement server push technology, we need the ability to send multiple parts of a single document using the HTTP/1.0 protocol. However *APACHE_1.2.5* has a bug where the server cannot be modified to keep multiple connections open and efficiently close them at the user's request. The details of the bug can be found in *APACHE's* bug reports.

5.8 An Example Site

In order to clarify the design in THTML server, a simple example site is designed and implemented. The experiment site is about the horoscopes. The time of clients' request determines the corresponding page to be shown. Each of them also have versions having the picture of the sign, or not, the user can also change the background of the page according to their preference.

Each page is linked with the previous sign page and the next sign page, so that the client can iterate all of the sign pages from either of them. The page combines temporal and user-defined HTML together. One can see how THTML creates a *dynamic site*. With conventional HTML, if each page has three versions of pictures and three versions of background, the total version will be 3*3 pages, each has the storage equal to the combined ones of the background and picture version.

While in THTML, *3+3* files are needed for the same amount of versions of the same site. The storage of the file is the same as each of them, as seen above, THTML save

huge storage in versioned site compared with conventional HTML.

In the above THTML, the client will see different pages of horoscope depending on the time of retrieving the page. The time accords to the time of horoscope at that time. In it, the request of previous and next one are also implemented with requests for particular temporal instances. The following two screenshots are a THTML source file and the corresponding conventional HTML page in response at a particular time. From this example, one can see how to create a temporal site in THTML.

```
<HTML>
<HEAD>
        <TITLE>Capricorn information - All about Capricorn!</TITLE>
<!-- Changed by: , 13-Jan-2000 -->
</HEAD>
<BODY background="Capricorn information - All about Capricorn!_files/whtmarb.gif">
<CENTER><IMG alt="capricorn" border="0" height="121"
src="Capricorn information - All about Capricorn!_files/capricorn.gif" width="161">
</CENTER>
<H1>
<CENTER><BR>The Goat<BR>December 22 to January 20</CENTER></H1>
<FONT color="#008000" size="+2"><B>
<HR>
<TABLE border="1" cellPadding="3" cellSpacing="3">
 <TBODY>
 <TR>
<TH align="middle" vAlign="top">
 <A href="home.M1IDnCgOxjcq@JaaervXx9eh-ebwm3F9VSA_6FXEwgae.html">
 <IMG border="0" src="Capricorn information - All about Capricorn!_files/button2.gif">
 <BR> Last Sign
        </A> </TH>
  <TH align="middle" vAlign="top">
 <A href="home.M1IDnCgOxjcq@JaaervXzJOXDQDaF7VLs4jFRnIlai.html">
<IMG border="0" src="Capricorn information - All about Capricorn!_files/button2.gif">
 <BR> Next Sign
   </A> </TH>
</U></TR>
</TABLE></FONT>

</BODY></HTML>
```

Figure 5.5 An Extension Sent to a Client at the Request Time 10:10:58 am, Jan 14, 00


```
<cron>
  * * * 22-20 12-1 *
</cron>
<HTML>
<HEAD>
<TITLE>Capricorn information - All about Capricorn!</TITLE>
<!-- Changed by:, 13-Jan-2000 -->
</HEAD>
<BODY background="Capricorn information - All about Capricorn!_files/whitmarb.gif">
<CENTER><IMG alt="capricorn" border="0" height="121"
src="Capricorn information - All about Capricorn!_files/capricorn.gif" width="161">
</CENTER>
<H1>
<CENTER><BR>The Goat<BR>December 22 to January 20</CENTER></H1>
<FONT color="#008000" size="+2"><B>
<HR>
<TABLE border="1" cellPadding="3" cellSpacing="3">
 <TBODY>
 <TR>
<TH align="middle" vAlign="top">
 <A href="home.html" TMOD="$Fri Nov 23 22:22:22">
 <IMG border="0" src="Capricorn information - All about Capricorn!_files/button2.gif">
 <BR> Last Sign </A> </TH>
   <TH align="middle" vAlign="top">
 <A href="home.html" TMOD="$Fri Jan 21 22:22:22">
<IMG border="0" src="Capricorn information - All about Capricorn!_files/button2.gif">
 <BR> Next Sign </A> </TH>
</U></TR>
 </TABLE></FONT>
</BODY></HTML>
```

Figure 5.6 The THTML Source for the Generated HTML Page Shown in Figure 5.5

Chapter 6

Conclusion and Further work

6.1 THTML----Dynamic HTML

There are two ways in which HTML pages can be dynamic. One method is to have the client's browser dynamically interpret an html file. DHTML belongs to this category. In DHTML, dynamic features can be implemented with functions written in JavaScript that can be interpreted and run by the clients' browsers. The other method is to have the server continually generate HTML files on the fly and sending them to the client. Examples of this method include Microsoft's ASP and XML. THTML introduced in this thesis also belongs to this category.

A THTML server behaves differently than an HTML server.

First, it provides temporal instances of a URL at a particular time context. It is able to do this by first, providing a system to store and retrieve the most relevant time-stamped HTML sources. Second, it also allows the client to embed time sensitive tags into conventional HTML components. Third, with push technology incorporated, THTML sites are updated using user-defined frequencies and preferences.

The above features are based on two major aspects in design. One is the TIS design in a temporal HTML file; the other is the refinement ordering of the time intervals.

In the former, the TIS is the time that a web document is expected to post. THTML combines the two data types in its representation: *date* and *time* to cover the required granularity. The format is as follows:

```
second minute hour day month year [dow]
```

With the above data type, basic time patterns in web authors' demands can be represented, which includes *before, after, during, time point* and *recurring* pattern.

In the latter, the extensional and intensional rules are specified in order to minimize

the ambiguity that may occur if multiple time intervals satisfy the time point in a client's request. The extensional rule specifies that the smaller time gap of the time interval stamp to a particular time point, the better-fit of the THTML source file it associates with. The intensional rule further refines the extensional rule to help resolve any ambiguities. The refinement order of the five time patterns that THTML supports is (from the lowest to the highest) *recurring, before/after, during, time point.* Among them, *before* and *after* have the same priority. Consequently, refinement ordering is not only related to the temporal gap but also the type of intervals. With the above algorithm, the ambiguities involved in finding the best-fit interval can be minimized. Therefore, the instance as server's response is generated using the THTML file whose time interval stamp fits the time point in request best.

6.2 Further Implementations

THTML was designed to fulfill more complex temporal requirements at the front-end. However some of these features still need to be implemented. They are:

- Server push technology needs to be integrated into the system. The server push interface needs to be enhanced with more options like allowing an alarm and pop up window when the site is updated. It should also allow the user to define the frequency of the updates.
- The file system can be isolated from the file server and developed into a large file pool. If temporal files are perceived as temporal data, all transactions can be perceived as storing data in one big repository and allowing the queries to retrieve the needed data from this repository.
- Natural language can be used in place of the time sensitive tags to make it more user friendly. In order to support the new features at the front end, another intensional tier can be incorporated at the back-end. This may include complex reductive rules and mathematical expressions to support more complex temporal requirements at the front-end. For example, if the author intends to post a page with a complex time pattern according to the Fibonacci function based on the two initial time points. With the Fibonacci rules incorporated at the backend, THTML is expected to fulfill the

requirement.

6.3 Possibility of Combining THTML, DHTML and XML

Although the three enhancements of HTML provide *dynamic* solutions from different perspectives, due to the fact that they are all based on hierarchical web document philosophy, there is possibility of combining them in use.

- With Java and JavaScript

Java is an object-oriented programming language that can be run on Web. It is embedded in HTML file, the browser finds the Java class file in the server and runs the program with Java Virtual machine installed.

JavaScript is an object-oriented scripting language embedded in HTML file. By defining

RUNAT tag, the browser can run the script at client side or server side. One enhancement is making it more interactive with callback functions.

DHTML perceives any part of the content as an object, and allows interaction and behavior of each with JavaScript and CCS.

THTML is the versioning solution of a web site with time With DHTML combined, any part of the web content can be distinct as a temporal versioned object (web elements). It will make the definition of web document beyond the limitation of time and content, which will give the web author more flexibility in designing web documents.

A Java and the object including server side JavaScript (Serverlet) can also be made temporal versioned entities. Based on different time, different Java program will be triggered at server side. (The author has no clue of how to combine the client-site JavaScript with THTML)

- With *XML*

XML allows user-defined tags in HTML file. There is a XML parser to parse it. Based on the particular data in particular user-defined tag, particular business logic is adopted at web server. From this aspect, it is similar to THTML. If THTML is

combined, XML elements can have multiple versions in time, different versions can be stored at server side. By combining the two, the user can write a document with user-defined styles as well as user-defined frequencies.

6.4 Prospect

One can see that as time-sensitive information on the web becomes more prevalent, the greater the need for adoption of THTML for such applications as newspaper and stock market sites. It allows the web readers to retrieve a version with a particular time. It also facilitates the planning of a site that changes very frequently without the user having to update it with the same frequency. With further implementation and the combination with the new web technologies, THTML will become a powerful authoring tool for *dynamic* web sites in advanced web applications.

Bibliography

[1] W.W.Wadge, "Possible wOOrlds", in Mehmet A. Orgun, Edward A. Ashcroft, editors, Intensional Programming 1, pages 56-62. Singapore: World Scientific, 1996.

[2] NCSA server documentation, http://hoohoo.ncsa.uiuc.edu, 1995

[3] W.W.Wadge and Taner Yildirim, "Intensional HTML", in Bill Wadge, editors, Proceedings of the Tenth Intensional Symposium on Languages for Intensional Programming", Victoria, B.C. Canada, 1997, pp.34-40

[4] J.A.Plaice and W.W.Wadge, "A new approach to version control", IEEE Transactions on Software Engineering, March 1993, pp. 268-276.

[5] Walter F. Tichy, "RCS- A system for version control", Software-Practice and Experience, vol. 15, no 7, 1985, pp.637-654

[6] W.W.Wadge and Edward A.Ashcroft, "Lucid, the Dataflow Programming Language", Academic Press Inc. 1985

[7] Mehmet A. Orgun and Weichang Du, "Multidimensional Logic Programming: Theoretical Foundations." Theoretical Computer Science, Vol185, 1997, pp319-345

[8] Avigdor Gal, Opher Etzion, "Extended Update Functionality in Temporal Databases", Temporal Databases, 1997, Dagstuhl, pp 56-95

[9] Robert Thau, "Design considerations for the Apache Server API", Fifth International World Web Conference,

http://www5conf.inria.fr/fich_html/papers/P20/Overview.html

[10] Gord Brown, "Intensional HTML 2: a practical approach, Master thesis 1998

[11] Ari Luotonen, "Web Proxy Servers", Prentice Hall, 1997.

[12] Yijun Lu, "Concept Hierarchy in Data Mining: Specification, Generation and Implementation", Master thesis of Simon Fraser University, 1997, pp33-59.

[13] Mehmet A. Orgun, William W. Wadge, "Extending Temporal Logic Programming with Choice Predicates Non-Determinism." Journal of Logic and Computation Vol 4, 1994, pp 877-903

[14] Chuchang Liu, Mehmet A. Orgun, "Dealing with Multiple Granularity of Time in Temporal Logic Programming", Journal of Symbolic Computation, Vol22, 1996, pp

699-720

[15] Mehmet A. Orgun, William W. Wadge: A Relational Algebra as a Query Language for Temporal DATALOG. Database and Experts Systems Applications, 1992, pp 276-281

[16] Opher Etzion, Avigdor Gal, Arie Segev, "Data Driven and Temporal Rules in PARDES", Rules in Database Systems 1993, pp 92-108

[17] Apache server documentation, http://www.apache.org, 1995

[18] W.W.Wadge, "Intensional Logic in Context", World Scientific, November 23, 1999

[19] Michael Bohlen, "Managing Temporal Knowledge in Deductive Databases", PhD thesis, 1995, Swiss Federal Institute of Technology