AN ABSTRACT OF THE THESIS OF

Dianne Hackborn for the degree of Master of Science in Computer Science presented on January 13, 1997. Title: Interactive HTML.

Abstract approved: ______________________________

Cherri Pancake

As the World Wide Web continues to grow, people clearly want to do much more with it than just publish static pages of text and graphics. While such increased interactivity has traditionally been accomplished through the use of server-side CGI scripts, much recent research on Web browsers has been on extending their capabilities through the addition of various types of client-side services. The most popular of these extensions take the form of plug-ins, applets, and "document scripts" such as JavaScript. However, because these extensions have been created in a haphazard way by a variety of independent groups, they suffer greatly in terms of flexibility, uniformity, and interoperability. Interactive HTML is a system that addresses these problems by combining plug-ins, applets, and document scripts into one uniform and cohesive architecture. It is implemented as an external C library that can be used by a browser programmer to add client-side services to the browser. The IHTML services are implemented as dynamically loaded "language modules," allowing new plug-ins and language interpreters to be added to an iHTML browser without recompiling the browser itself. The system is currently integrated with NCSA's X Mosaic browser and includes language modules for a text viewer plug-in and Python language interpreter. This thesis examines the iHTML architecture in the context of the historical development of Web client-side services and presents an example of iHTML's use to collect usage information about Web documents.

Interactive HTML

by

Dianne Hackborn

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed January 13, 1997
Commencement June 1997

Master of Science thesis of Dianne Hackborn presented on January 13, 1997

APPROVED:

Major Professor, representing Computer Science

Chair of the Department of Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Dianne Hackborn, Author

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

Table | Page

INTERACTIVE HTML

1. INTRODUCTION

The World Wide Web is an Internet-based distributed hypertext system. Resources on the Web are described by character-strings called Uniform Resource Locators (URLs) [5], which encode information about the machine and directory at which the resource exists, and the network protocol used to retrieve it. These resources are usually retrieved by the core protocol of the Web, the Hypertext Transfer Protocol (HTTP) [4], which is similar to the traditional FTP protocol except that it encapsulates the returned data in a MIME [7] header, supplying meta-information about it such as the type of document it is (e.g., GIF graphics, postscript, WAV audio). The user agent used to access the Web is commonly referred to a "web browser." Every web browser can display the core Web document format, the Hypertext Markup Language (HTML) [3], which is an SGML-based [20] format for encoding textual information and hyperlinks. Graphical browser cans usually display images that are embedded in HTML documents, and there are a large number of other types of files in use on the Web that modern browsers need to be able to handle.

Interactive HTML (iHTML) is a system for adding various client-side services to web browsers. Its basic architecture is designed to be browser-independent, so that its capabilities will work the same way across a variety of browsers; in addition, it attempts to make use of currently developing Web standards—HTML [3], HTTP [4], and other existing Web protocols—that help support these client-side services.

1.1. Motivation

As the Web continues to grow, people clearly want to do much more with it than just publish static pages of text and graphics. Consequently, much of the recent research on Web browsers has been on extending the capabilities of browsers through the addition of various types of client-side services. This section will first examine some of these services, then discuss the problems that currently exist in their practical use.

1.1.1. Current Service Structure

There are three general types of client-side services that have become widely deployed in Web browsers: plug-ins, applets, and document scripts.

Plug-ins are compiled code modules stored on the same machine as the browser, that work in cooperation with the browser to implement new capabilities. When the browser encounters a feature that it doesn't understand—typically a file format such as an animation, sound, or 3D model—or for which it doesn't have a built-in handler, it looks for a plug-in that provides that capability. If one is found, the browser loads and executes the plug-in. The two elements thus work together to implement the new feature.

Applets are architecture-independent pieces of code that a browser retrieves from the network and executes on its local machine in order to implement new features. Conceptually, applets are very similar to plug-ins, except that they are presented to the browser at a higher level than the raw machine code of plug-ins, allowing them to be executed on any computer architecture or operating system. In addition, since they are retrieved from the net, their access to the local machine must be limited to a "safe" subset of operations. Applets may be written in a variety of languages, but a browser must support the particular language used by an applet in order for it to work.

Document Scripts are architecture-independent pieces of code that work with the browser to control the display of, and user interaction with, an entire document. A plug-in or applet exists in an environment apart from the document in which it appears; it is given a portion of the display, and can use that area to interact with the user, without regard for whatever the surrounding document is. In contrast, document scripts are fundamentally concerned with the document itself, providing services such as updating one form element when the user modifies another, dynamically creating parts of a document, responding to the user clicking on a hyperlink in the document, etc.

In addition, there is an unnecessary distinction made by plug-ins and applets between a "program" and the other data types that are retrieved from the Web. For example, plug-ins were designed almost entirely as a mechanism to display new types of "data," while Java and JavaScript were built into browsers to execute programs. But there's no need to view a program retrieved from the net as anything different than a picture or sound. They can both be handled with a plug-in, provided an appropriate interface to the browser is in place.

1.1.2. Limitations of Existing Service Structure

The previously described client-side services were created in a haphazard way by a variety of independent groups. As a result, they suffer greatly in terms of flexibility and uniformity. There are two basic issues: the variations and idiosyncracies within each type of service, and a lack of interoperability among different types of services.

The most important example of variation within a client-side service type is in the realm of the applet. Here, the most widely deployed applet implementation is Sun's Java [18]. Java executes a particular virtual machine [26] and its own higher-level API [17], but there is substantial evidence that Java cannot fill the role of a general applet language. (In the next chapter, a number of experimental browsers supporting other applet languages are described, each demonstrating the unique features that language provides to support a particular problem domain.) An example of this can be seen by comparing Java with Tcl/Tk. Java is similar to C or C++ in design—it's a compiled language, with strong type checking. While it does provide some features that help in rapid prototyping (garbage collection being the most important) this is really not its primary focus or strong programming domain. Tcl/Tk, on the other hand, is designed for the express purpose of rapid prototyping. Considering the extremely dynamic nature of the Web, it seems likely that rapid prototyping will be a fairly common endeavor, but for this Java makes a very poor substitute.

One commonly suggested solution to the need for a different type of language support is to compile another language to the level of Java's own virtual machine. (Indeed, there exists a Tcl interpreter written in Java.) However, a large part of what makes Tcl/Tk so useful is the high-level widgets that are supplied by it, and no such support is available in Java. There are also other competing virtual machines, such as Lucent Technology's Inferno [10], that present their own advantages and disadvantages with regard to Java.

Another solution may be to write the other languages as browser plug-ins. Plug-ins theoretically *should* allow any scripting language to be used in a browser. In practice, however, this is not the case. Plug-ins were designed to be used primarily for new *media* types, and thus do not have the kind of interface to the browser (such as the ability to retrieve parameters) that is needed to implement a real scripting language like Java.

This leads to the second important issue: the lack of interoperability *among* service types. For the most part, each service type exists in its own world. A plug-in, for example, provides a very different set of capabilities and possible interactions with the browser than does a document script or applet. The use of Java and JavaScript in Netscape is a dramatic illustration. Java is used to write applets, while JavaScript is used to write document scripts. Therefore, the kinds of things that can be done in these languages are very different. There is no way to write a Java program that generates part of an HTML document or controls forms, features that can be accomplished easily in JavaScript. Conversely, graphical applets cannot be created with JavaScript. Yet there are good reasons to want to do both of these things.

In spite of its name, JavaScript is nothing like Java. Rather, it is a high-level interpreted language—just the kind of language that would be useful for the rapid prototyping of applets. Similarly, the more efficient nature of Java, which is compiled, would be very useful for writing document scripts that must do a lot of computation. Netscape has tried to fix these limitation by introducing LiveScript, which allows Java and JavaScript programs to interact with each other and plug-ins. This does not get at the core problem of integration, however, and leaves the programmer in the position of having to understand two or more languages to accomplish a task.

1.2. Interactive HTML

IHTML provides an architecture that combines the three client-side services discussed in the previous section into one integrated environment. This integration is done in two steps: by first combining the plug-in and applet services, then incorporating document scripting services into that result.

Plug-ins and applets are combined by taking a cue from the World Wide Web Consortium's recent `<OBJECT>` tag [22] specification. The idea is to consider normal data types (e.g., images, sounds, text) as being no different than scripts. To the browser they are all simply "some piece of data that I need to know how to show to the user." The `<OBJECT>` tag provides most of the HTML syntax needed to do this. All iHTML must do to complete the picture is to provide an interface between the browser and the plug-in that allows the latter to handle all data types, including programs. Consider, for example, the common desire to embed some new image format inside a document. Plug-ins and applets provide two ways through which this can be accomplished: by finding (or writing) a plug-in that will display the data, or writing an applet that can display it. Both of these have disadvantages, however. Using a plug-in requires that the user have this plug-in installed and available, while applets are interpreted and thus execute a lot slower than the equivalent plug-in. The combination of these two mechanisms, however, can greatly reduce their disadvantages. Figure 1.1 shows an example of this. Here, an MPEG animation is being embedded into a document. If the browser has a plug-in that can display this type, it is used. However, if it doesn't, it retrieves the applet "plaympeg.py" and uses that to display it (assuming, of course, that there is a Python language plug-in available). Though the applet won't be able to display the animation as well, the user will still be able to see something even in the absence of a local MPEG player.

```
<OBJECT data="myanim.mpg" type="video/mpeg"
        classid="plaympeg.py"
        classtype="application/x-ihtml; language=python">
</OBJECT>
```

Figure 1.1 Combining plug-ins and applets

The second step, combining this new type of plug-in with document script services, basically requires that the plug-in interface be expanded yet again to allow the plug-in to interact with the browser in the same ways that a document script can. Because of the close connection between browser and document script, this requires careful planning so that the interface does not become dependent on a particular browser implementation. Some additional HTML syntax is also necessary so that these new document scripts can work as plug-ins. By bringing document scripts under the same general system as plug-ins, these scripts can now be written in the same language that the programmer is familiar with in plug-ins, or other languages that have specific strengths for a particular application. For example, a document script that dynamically checks the validity of a form with a server-side database can now be written in Java. Doing this allows the programmer to open an interactive network session with the database, and take advantage of the superior computational capabilities of Java.

1.3. Users

Because the people using iHTML represent a very wide variety of needs and interests, to help delineate the interests of its users we break them down into five

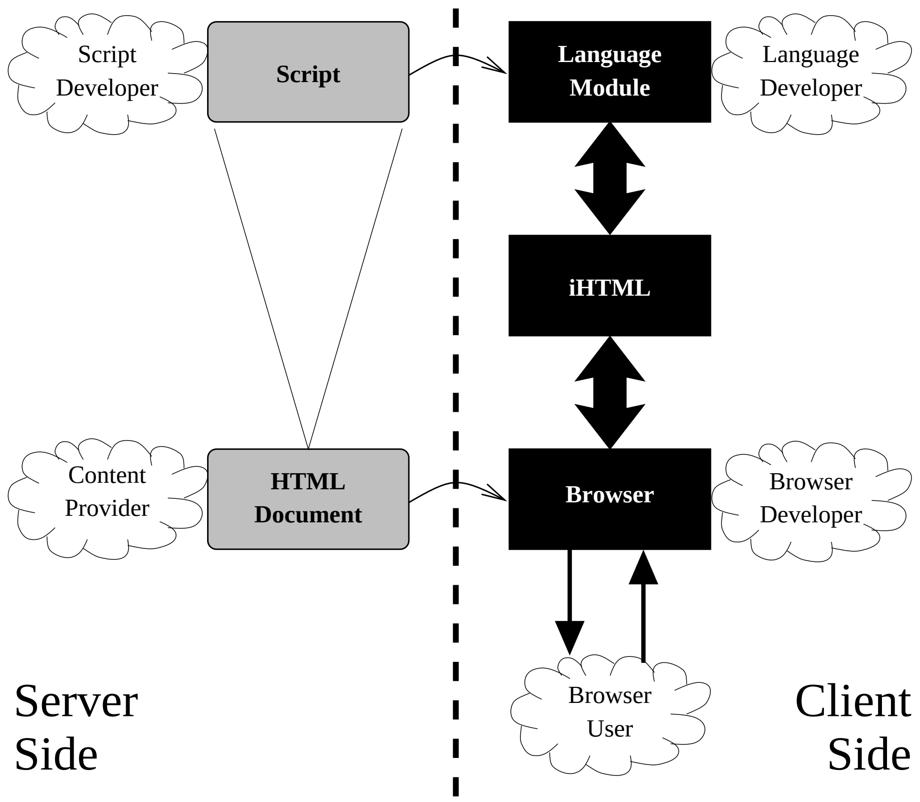


Figure 1.2 The users of the iHTML system

explicit categories, as shown in Figure 1.2. This figure shows the various parts of the iHTML architecture, and the user category that is directly involved with each.

1. A **browser developer** is a person who has implemented a browser, such as Netscape or Mosaic, or adapted an existing browser, so that it supports the iHTML interface.

2. A **language developer** is a person who has implemented a language (or is taking someone else's language implementation) and is writing the code to interface it with the iHTML system. This user category also includes current plug-in developers, though these people will not normally use many of the new capabilities introduced by iHTML.

3. A **script developer** writes the actual applet or other piece of client-side code, using the language supplied by a **language developer**, and intended to be incorporated into an actual HTML document created by a **content provider**.

4. A **content provider** is a person who creates HTML documents and other information and makes it available on the Web, possibly incorporating client-side scripts created by a **script developer**.

5. A **browser user** is a person who is running a browser written by a **browser developer** to view the documents created by a **content provider**.

1.4. Guide to This Thesis

This chapter has presented a brief introduction to the iHTML system and problems it attempts to address. Chapter 2 will go into further detail on the existing client side services used in browsers, as well as some of the methods of collecting Web usage statistics. Chapter 3 then provides a more detailed description of the general iHTML architecture and concepts it uses. Chapter 4 presents a detailed description of the script developer's view of the Python language module, with Chapter 5 providing a detailed analysis of example Python scripts used to collect and display document usage information. To provide further detail on the underlying iHTML system, Chapter 6 describes the system from the perspective of the language developer, then Chapter 7 delves into the implementation details of the system. Finally, Chapter 8 presents some final concluding thoughts on the contributions and limitations of the system.

2. RELATED WORK

This chapter presents an overview of the research that has been done on the various types of browser client-side services with which iHTML is concerned. It first discusses the existing research on plug-ins, applets, and document scripts, then examines the work that has been done on integrating these three types of services. Finally, it looks at some of the effort that has gone into collecting usage information on Web pages, and research in other areas of computer science that may indicate additional kinds of usage information about Web pages that would be useful to collect.

2.1. Plug-ins

The plug-in was the first kind of client-side service that was integrated into Web browsers. These systems generally consist of two pieces:

1. The browser itself, with a publicly defined programming interface that external pieces of code can make use of to add new features to the core browser.

2. A set of code modules that implement these new features; the browser will typically load and execute these modules as they are needed.

Plug-ins are almost always used to add new data type handling to the browser. To do this, the browser has some list of mappings from the plug-ins themselves to the data types they handle. When the browser encounters a data type that it doesn't understand, it searches through this list for a plug-in that can handle it; if found, the browser and plug-in work together to display the data.

Note that the data types plug-ins can handle are not limited to graphical formats. Because the plug-in is compiled machine code, it has full access to the system, allowing it, for example, to use the computer's audio hardware to play a sound sample.

2.1.1. Eòlas's Weblets

Martin *et al.* published the first example of embedding dynamic content into HTML documents using their "weblet" architecture, which appeared in a report on integrating a scientific visualization application with the Mosaic browser [28]. Soon after, they published a more detailed description of their architecture for integrating external applications with Web browsers [1]. These papers, which introduced the advantages of extending the capabilities of Web browsers, laid much of the foundation for later work in using client-side extensions to help off-load computation from servers.

There are two main elements to their system. The first is a syntax for indicating where dynamic content is to be embedded into an HTML document, and what that content is to be. For this, they defined a new HTML tag called `<EMBED>`, conceptually very similar to the standard `<IMG>` tag. That is, it indicates that a rectangular graphical area is to appear in the document, inserted within the HTML text defined with it. Instead of a static image, however, the reserved area will be drawn by an external application. The tag's syntax is shown in Figure 2.1 in the form of an SGML document type definition (DTD) [46]. The tag includes four attributes to control what is actually displayed, all of which must be supplied: `HREF` gives the URL from which to retrieve the data to display, `TYPE` indicates the type of this data as per the HTTP Content-type field [4], and `WIDTH` and `HEIGHT` supply the pixel dimensions of the display area. Figure 2.2 shows an example of the `<EMBED>` tag being used to embed an MPEG animation into a document.

```
<!ELEMENT   EMBED   EMPTY>
<!ATTLIST   EMBED
            HREF    %URL    #REQUIRED
            TYPE    CDATA   #REQUIRED
            WIDTH   NUMBER  #REQUIRED
            HEIGHT  NUMBER  #REQUIRED>
```

Figure 2.1 DTD of the Eòlas <EMBED> tag

```
<EMBED
  HREF="http://www.spa.com/water.mpg"
  TYPE="video/mpeg"
  WIDTH=400
  HEIGHT=400>
```

Figure 2.2 Example Eòlas weblet HTML to display an MPEG animation

The second element of this system is the mechanism used to connect the browser with the application to create an image. Upon encountering an <EMBED> tag, the browser looks for an external application that both supports the <EMBED> protocol and is capable of handling the given data type. If one is found, the browser allocates an off-screen pixmap of the size indicated by the tag, starts the external application as another process, and passes it a handle to the pixmap. The external application can then draw directly into the pixmap, sending a message to the browser when it has completed the image. When the browser receives this message, it copies the pixmap's current image into the rectangular area reserved in the document, thus updating the user's view. Since the external application has full access to the computer where the browser is running, it can take any steps necessary to handle non-graphical data types (such as sending a sound file to the computer's audio hardware).

This system offers essentially no support for interaction between the external application and the user via the image it shares with the browser. Because the authors' primary interest was in integrating existing scientific visualization packages with Web documents, their architecture assumes that the external program will open its own windows (outside of the Web browser) in order to support interaction with the user. The programming interface between the external application and browser is thus very simple. There are only three messages the application may send to the browser (indicating when it starts, stops, and has completed drawing an image); the browser may send another three messages to the application (telling it when its display area is visible or invisible, and informing it if the browser terminates). Of particular note is that there are no messages transmitted to the external application about user events intercepted by the browser.

The primary advantage of the weblet approach is that it allows a very simple interface between browser and external application, making it extremely easy to modify existing programs to work with the system. In many other domains, however, it would be useful to embed user interface mechanisms directly into the document, a feature that is not compatible with this design. In addition, this system assumes that the external application handling a particular data type will be resident on every client machine that wishes to view the data.

2.1.2. Netscape's Plug-ins

The next plug-in architecture to be documented, and the first to achieve widespread use, was Netscape's Plug-in API [39]. It is important primarily for its role in refining the plug-in concept. Rather than a complete external application that communicate with the browser through messages, the plug-in is a specialized code module that the browser executes within its own address space. This allows a plug-in to have direct access to its display area in the browser, allowing it to create an embedded interfaces that is much more interactive than the Eòlas weblet. It also moves the plug-in concept toward one of its important roles: off-loading processing from remote servers to local execution on client machines. In addition, the Netscape architecture expands type-handling, having the browser make use of the HTTP Content-type header when selecting a plug-in that it will use. Use of the file's embedded type information represents an important step toward the encapsulation of remote files as abstract objects.

As with the Eòlas weblet, there are two components to the Netscape plug-in architecture. The `<EMBED>` HTML tag is used again for specifying the plug-in in a document. This is very similar to the Eòlas tag, recognizing the `WIDTH` and `HEIGHT` attributes as the dimensions of the plug-in's display area. The `TYPE` attribute is now optional, however; if not supplied, the HTTP Content-type information is used instead. The `HREF` attribute has been re-named to `SRC` but, except for being optional, works the same way. New attributes include: `HIDDEN`, to indicate that the plug-in should not be displayed (useful, for example, for a sound plug-in); `PLUGINSPAGE`, giving the URL for a document with instructions on how the plug-in should be installed; and `PALETTE` and `UNITS` are additional refinements to the tag, controlling the plug-in's palette mode under Windows and screen units the dimensions are specified in, respectively.

The more important component, however, is the so-called Plug-in API. This is a C language interface between the plug-in and the browser itself. It defines a set of functions that the plug-in can use to control the browser's behavior, as well as functions that the plug-in must supply for the browser to use. Fourteen browser functions allow the plug-in to tell the browser to retrieve and display a new URL, post data to a URL (as happens when submitting an HTML `<FORM>`), allocate and free memory, display messages in the browser window, and dynamically retrieve information about the browser itself. Several of the functions are for manipulating what Netscape calls "streams;" these are byte-streams of data retrieved from an arbitrary URL. The plug-in can open a URL stream, and is then handed the data from that stream as it becomes available; this feature is most often used to retrieve the URL requested by the `SRC` attribute, in order to display it.

Another fourteen functions must be provided by the plug-in; these are called by the browser in order to send information to the plug-in. They include functions that inform the plug-in that it has been loaded or deleted, that instances of the plug-in are created or destroyed, and that the plug-in should print its current screen display. There are also several stream-related functions that the browser uses to inform the plug-in about stream creation or destruction, or to supply data as it is retrieved from an active stream. Another function passes the plug-in a list of the attributes supplied in its `<EMBED>` tag, organized as two arrays (one for the names of the attributes, one for their values).

The most important interaction between the plug-in and browser is not, however, defined directly by this API—how the plug-in is to display itself in the document and interact with the user. Instead, the browser calls a plug-in function through which it gives the plug-in an abstract handle to its display area; this handle is the operating system's actual sub-window that appears there (or, on operating systems that do not support sub-windows, the plug-in's physical coordinates in the browser's window). The plug-in can then make direct use of standard operating system calls to draw in that window and create any interface elements (buttons, text fields, etc.) it needs. It also receives user interface events through the window, just as it would when written as a stand-alone application.

While the Netscape system provides a very good model for a standard plug-in architecture, it has a few important weaknesses. The first is the syntax of its `<EMBED>` tag. Any number of arbitrary attributes may be included with the tag; if present, they are assumed to supply custom parameters to the plug-in itself. For example, `<EMBED SRC="anim.mpg" SPEED="1">` would supply the MPEG player with a "speed" parameter set to the value "1". This syntax, however, is not valid SGML, since any SGML DTD (which HTML is) requires that all tag attributes be explicitly specified. This means that a DTD can not be written to verify the correctness of an HTML document that includes the `<EMBED>` tag.

There are also problems in the underlying plug-in architecture. The most important is the limited access the plug-in has to the browser itself. There is no way for a plug-in to generate HTML code, access information about the document it is running on, or perform other operations that afford a closer integration between the plug-in and browser. This becomes an important consideration as other client-side methods are considered in the sections below. In addition, while making direct use of the underlying operating system's graphical support provides a large amount of control to the plug-in, it also makes it likely that the plug-in will be difficult to port to new computer platforms. Since a basic property of the Web is its multi-platform nature, encouraging the widest possible availability of plug-ins is an important issue.

2.2. Applets

Applets are, in many ways, similar to plug-ins: they are external pieces of code that the browser executes to add new capabilities and features. When in use, the applet itself is given a part of the document in which it can draw and interact with the user. Unlike plug-ins, however, applets are written in a machine-independent language. This means that one applet can, theoretically, work with any browser that needs it: the only requirement is that the browser understand the language it is written in. Thus, applets are usually retrieved from the network by the browser, as they are needed.

In a sense, applets can be thought as a special type of plug-in. The data type is the applet language, and this data is rendered by executing the program. They cannot usually be implemented as plug-ins, however, because the language needs a closer connection to the browser than that supported by existing plug-in architectures.

2.2.1. Sun's HotJava

Applets were first introduced by Sun's HotJava browser and its associated programming language, Java [18]. The Java language is object-oriented and closely modeled after C++. It is compiled into a machine-independent representation, referred to as byte-code, which is then interpreted on the client machine. The HotJava browser is itself written entirely in Java, and can download pieces of Java code from the network, executing them on the client machine just as if they were an integral part of the browser. These pieces of code can provide three types of extensions to the browser: **dynamic types**, for interpreting and displaying new types of data; **dynamic content**, for adding "interactive content" to a Web page; and **dynamic protocols**, for introducing new network protocols (i.e., a new URL scheme such as "`http:`" or "`news:`").

The most important of these, and the one corresponding to the traditional applet, is the concept of dynamic content. Dynamic types are very similar to the plug-ins previously discussed, except that their implementation in Java allows them to be platform independent (further discussed in Section 2.4.2). Dynamic protocols are not yet widely available, as their implementation is strongly tied to the underlying browser's implementation. Since they are not addressed by iHTML, they will not be discussed further.

In HotJava, dynamic content is introduced with the `<APPLET>` tag.[1] Unlike the tags discussed so far, `<APPLET>` is a container—additional text and tags may appear between it and its closing `</APPLET>` tag. The complete document type definition is shown in Figure 2.3. The tag is used to specify the location of an external program, which is retrieved over the network by the browser; the program executes on the local machine, within its own display area in the document. Arbitrary parameters can be passed to the applet with the `<PARAM>` tag, appearing as part of the `<APPLET>`'s content. Browsers that are unable to display the applet will display the other `<APPLET>` content, allowing documents to fall-back gracefully on older browsers that do not support applets.

The following is an example of using the tag to insert a scrolling banner in a document:

```
<APPLET CODE="banner.class" ALIGN=CENTER>
   <H3>Can't display banner applet.</H3>
   <PARAM NAME="text" VALUE="This is a banner applet.">
</APPLET>
```

The banner program is given one argument, "text", which tells it to scroll the text "This is a banner applet." If the browser does not support applets, it instead displays the heading "Can't display banner applet."

---

[1]The original version of HotJava used an early tag, named `<APP>`, with slightly different syntax. Its syntax was not a valid SGML construct, however, and was later replaced with the SGML-compliant `<APPLET>` syntax.

```
<!ELEMENT APPLET - - (%text)* +(PARAM)>
<!ATTLIST APPLET
  codebase %URL     #IMPLIED   -- code base --
  code     CDATA    #REQUIRED  -- class file --
  alt      CDATA    #IMPLIED   -- for display in place of applet --
  name     CDATA    #IMPLIED   -- applet name --
  width    %Pixels  #REQUIRED  -- suggested width in pixels --
  height   %Pixels  #REQUIRED  -- suggested height in pixels --
  align    %IAlign  #IMPLIED   -- vertical or horizontal alignment --
  hspace   %Pixels  #IMPLIED   -- suggested horizontal gutter --
  vspace   %Pixels  #IMPLIED   -- suggested vertical gutter --
  >

<!ELEMENT PARAM - O EMPTY>
<!ATTLIST PARAM
  name    NAME      #REQUIRED  -- The name of the parameter --
  value   CDATA     #IMPLIED   -- The value of the parameter --
  >
```

Figure 2.3 DTD of the HotJava `<APPLET>` tag (from the World Wide Web Consortium's HTML 3.2 specification [43])

As with Netscape's plug-in architecture, the HotJava applet has complete control over its display area. It can arbitrarily create its own user interface elements and has direct access to the user events associated with them. Unlike plug-ins, the applets are platform-independent, since they are written in the Java language [17] and executed in the form of virtual machine code [26]. Also, instead of directly accessing the underlying operating system, applets go through the Java Application Programming Interface [19] layer. Finally, Java applets are "safe" (they are not allowed to perform any operations that would compromise the local machine), so the browser can download them automatically; plug-ins, in contrast, need to be manually downloaded and installed by the user.

Originally, HotJava's applets were not portable between different browsers. Instead, they were closely tied to the browser itself, which was also written in Java. To execute an applet, the HotJava browser simply retrieved the needed code and executed it along with the rest of the browser, treating it no differently (beyond applying the standard Java security restrictions). Browsers that are not written in the Java language can not directly execute them in this way, however.

2.2.2. Applets in Other Languages

Soon after the HotJava browser was announced, a number of other browsers were created that demonstrated how the concepts it introduced could be implemented in other languages. These showed the utility of having a variety of language options and approaches available.

Tcl/Tk [40] is a high-level, string-based, extensible scripting language and user interface widget set based on the X Window System. The Tcl language itself has often been suggested as a good language for transmitting programs over a network because it is interpreted, portable, and can provide a "secure" execution environment. The Tk widget set adds a powerful set of high-level user interface elements, making the system very useful for rapid prototyping and creating the smaller-scale programs that are common in applets. Tcl/Tk's author (John Ousterhout at Berkeley) has said:

> Tcl/Tk and Java are siblings in a family of products for programming the Internet. They don't conflict with each other any more than Tcl conflicts with C or C++. Each language has particular strengths, and developers will choose one or the other or both, depending on the needs of their applications. We think that this two-language approach will

> provide developers with much more flexibility and power than any single-language approach. [41]

There are currently two systems featuring Tcl/Tk applets. The first available was the SurfIt! browser [2], which was completely written in Tcl/Tk. More recently, Sun has released a Tcl/Tk plug-in [25] for the Netscape browser.

Python [27] is a language similar to Tcl/Tk that has also been used for creating applets. Like Tcl/Tk, it is interpreted, portable, and provides a safe execution environment. However, it is also a fully object-oriented language, with a much more sophisticated concept of scoping and types. and therefore is more appropriate for medium and large prototypes than Tcl/Tk. Its speed is somewhere between Java and Tcl/Tk, making it extremely useful for developing larger applets where execution speed is not of primary importance. While such applets are also possible in Java, they are in many cases smaller and easier to develop in Python. To this end, the Grail browser [49] was developed for creating Python applets and is itself written in Python. Grail's demo applets include a number of the original Java demos re-written as only slightly slower Python applet. These demos highlight the simplicity and rapid prototyping capabilities of the language while demonstrating that it still executes fast enough to address some of the domains that Java is often used for.

In contrast to the languages mentioned so far, Caml Special Light is a functional language. It is a statically typed compiled language. Applets are sent over the network in the form of byte code and execution safety is ensured through byte code validation and PGP signatures. The characteristics of functional programs—particularly the elimination of run-time type errors—suggests that they can provide more efficient code generation than Java. The showcase for Caml applets is the MMM browser [8], which is also written in the Caml language.

The languages and browsers discussed so far are similar to Java in that the applet language matches the language its browser is written in. This ties the two closely together. For example, an applet typically uses the same user interface architecture as the browser, so the interface between the two is usually implemented by giving the applet a direct handle on the display element that the browser is using to represent it. This interface breaks, however, when an attempt is made to move these applets to other browsers written with different user interface libraries or in different languages.

There are also a number of proposals for implementing applets that look at the underlying machine representation instead of focussing on a particular language. For example, the Modula-3 byte code used in the Visual Obliq browser [6] presents a possibly more efficient and mature architecture to Java's byte code. Another system, Juice [15], uses Oberon's intermediate parse-tree representation to generate code dynamically on the host machine [14]. This leads to smaller code and much quicker execution than an equivalent Java program.

2.2.3. The Java Platform

The Java Platform [23] describes a subset of the Java applet architecture described in Section 2.2.1 that can be implemented by browsers that are not themselves written in the Java language. This allows plug-ins to remain independent not only of the hardware on which they are running, but also of any particular browser implementation. In order to accomplish this, the extensibility of the browser must be limited to the well-defined applet interface. Other extensions that require more direct access to the internal browser implementation (such as dynamic protocols) are not supported.

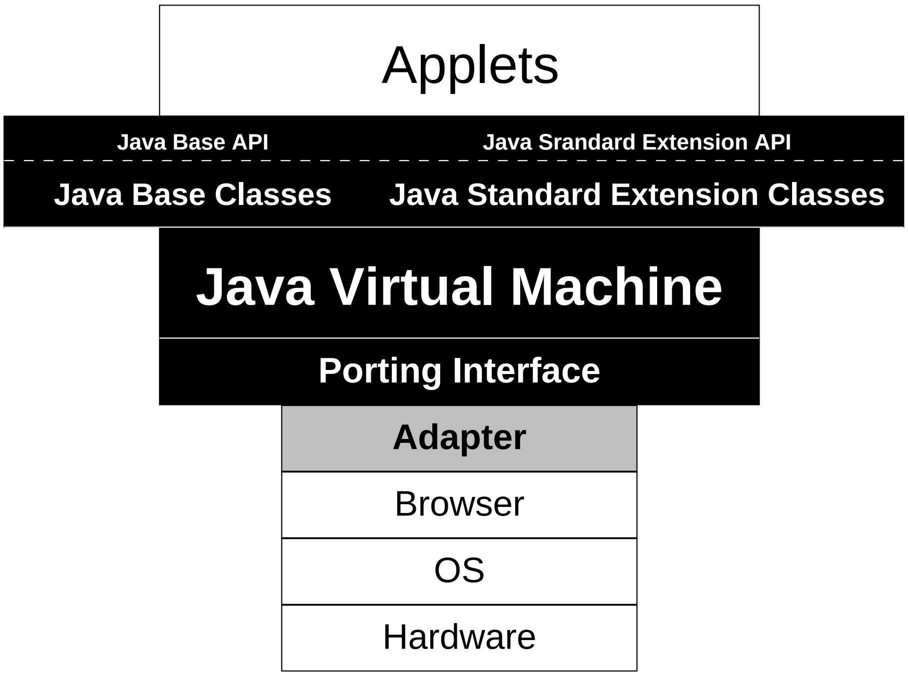


Figure 2.4 The Java Platform Architecture for browsers (adapted from [23])

There are three parts to this system, as illustrated in Figure 2.4. First, the Java interpreter is encapsulated in its own component that exists apart from the browser itself (represented by the boxes in black). It implements all the Java-specific parts of the applet—the Java virtual machine [26] and application programmer interface [17]. Second is the browser itself and the underlying operating system (represented by the white boxes). This can be implemented in any language and architecture. The third component connects the first two, allowing them to communicate and interact with each other (represented by the gray box).

While this architecture greatly improves the ability to widely deploy applets as a general part of Web browsers, it has a number of limitations. The most important of these is its dependency on Java. As previously mentioned (Section 2.2.2), there are many other languages that supply their own unique advantages and strengths in the implementation of applets, and many more will likely be developed in the future. Leaving these out from wider use is a serious weakness. In addition, the Java interpreter here is tightly bound to the browser itself, being compiled directly into the core executable. This makes it impossible to take advantage of improvements to the language without updating the browser. It also adds unnecessary overhead when the language is not being used, and thus does not scale well to situations in which more than one applet language is available.

2.3. Document Scripts

The final type of client-side service that will be discussed is the document script. Like an applet, a document script is an architecture-independent program that a browser retrieves from the network and executes on its local machine. Rather than executing as an element embedded in the document, however, a document script works hand-in-hand with the document. It can exercise control over how it is presented to the browser user and how the user interacts with it (for example, it can dynamically generate parts of the document, control form input, etc.). The document script thus has an intimate tie to the browser itself, since such programs need access to much of the internal information (such as form state, link information, etc.) that a browser has about the document.

2.3.1. The Common Client Interface

The National Center for Supercomputing Application's Common Client Interface (CCI) [33] was the first document-level programming interface to a Web browser. It was originally designed to provide external applications with remote control of a Web browser, for implementing such things as slide-shows, document sharing systems, and teacher/student interactions.

The CCI architecture is socket-based; that is, the interface between the Web browser and external application is through a standard TCP/IP socket. Four steps are required to make use of the protocol:

1. The browser user selects an arbitrary port on which the browser will begin listening for socket connections.

2. The browser user sets the external application to use this same port number.

3. The external application opens a TCP/IP socket on that port, establishing a connection with the browser.

4. The external application sends commands to the browser, and asks for reports from it on various types of information.

Thus, the protocol is based on messages sent back and forth over the socket such that, in general, the browser operates as a slave to the external application, responding to commands sent by it and reporting various types of events and data it asks for.

On top of the socket-level protocol is a library with a C-language interface; generally, there is a fairly direct mapping between the C functions and underlying socket messages. By invoking this library, the external application can check on the status of the connection (e.g., if the connection is still active or if there is data available from the browser), instruct the browser to fetch and display a new URL, and execute HTTP "POST" operations through the browser. In addition, through a callback mechanism the external application may be notified when the user selects anchors, may intercept documents retrieved by the browser based on their MIME types (e.g., tell the browser to send all "application/postscript" documents it retrieves over to itself instead of letting the browser display them), and get information about the structure of the document currently visible in the browser.

One important weakness of the CCI protocol is that it is not generally automatic. The browser user must configure the communication ports in both the browser and external application before that application can take control. In addition, it is difficult to associate a page (or set of pages) with an external application; for example, there is no way to write a CCI program that is linked to a document so that it automatically begins execution when the document is retrieved. In many ways, this protocol is oriented more towards browser-level than document-level operations.

2.3.2. Netscape's JavaScript

Netscape's "JavaScript" [37] is the first true example of a document scripting architecture. JavaScript (which is not related, in any beyond name, to Sun's Java) is a custom language that executes inside a Web browser. Programs written in the language execute in the context of an entire HTML document, allowing them to do such things as dynamically construct a portion of a document, interact with the user through forms, load new documents, etc. The language is interpreted, so it is machine independent.

Unlike other client-side scripts, JavaScript programs are traditionally embedded directly into an HTML document. The embedding is accomplished with the `<SCRIPT>` tag, the DTD for which is shown in Figure 2.5. The tag's two key attributes are `LANGUAGE` and `SRC`. The former is the name of the scripting language, which must always be "`JavaScript`", while the latter specifies an optional remote

```
<!ELEMENT SCRIPT - - CDATA -- script statements -->
<!ATTLIST SCRIPT
  type     CDATA  #IMPLIED -- Media type for style notation --
  language CDATA #IMPLIED -- predefined script language name --
  src      %URL   #IMPLIED -- URL for an external script --
    >
```

Figure 2.5 DTD of the JavaScript `<SCRIPT>` tag (from the World Wide Web Consortium's HTML 3.2 specification [43])

URL from which to retrieve the script (discussed further below). The script contained in a `<SCRIPT>` tag is executed as soon as it is encountered in the document. One of the most important things that it can do is to construct dynamically parts of the document:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
document.write("Hello net.")
</SCRIPT>
</HEAD>
<BODY>
That's all, folks.
</BODY>
</HTML>
```

This example (from the JavaScript Authoring Guide [37]) results in a document with the text "`Hello net.  That's all, folks.`"

The second important capability of JavaScript is that of responding to user interface events occurring within a document (e.g., editing form elements, selecting links, etc.). To support this, JavaScript adds new attributes to the various HTML tags at which events may appear. An example is the following fragment of the DTD for the `<A>` tag (as specified by the World Wide Web Consortium's HTML 3.2 specification [43]):

```
<!ELEMENT A - - (%text)* -(A)>
<!ATTLIST A
        %attrs;     -- id, class, style, lang, dir --
        ...
        onClick     %script #IMPLIED  -- intrinsic event --
        onMouseOver %script #IMPLIED  -- intrinsic event --
        onMouseOut  %script #IMPLIED  -- intrinsic event --
        >
```

Three events are available for the `<A>` tag. The value of these attributes is an actual fragment of JavaScript code that will be interpreted when the event occurs. Typically, this code is simply a call to a function that was defined in an earlier `<SCRIPT>` block. Figure 2.6 illustrates this concept with an example script that would respond to click events on a form button.

There are a number of important problems with JavaScript. The first, and most overt, is simply its HTML syntax. Embedding these scripts directly into a document not only blurs the important separation between a document's *content* and its *implementation*, but also represents invalid SGML. A tag with content type CDATA, as `<SCRIPT>` is declared, is essentially terminated by "anything that looks like an end tag" [13], meaning that JavaScript programs with statements such as "`document.write("<B>bold</B>")`" are invalid. In general, the use of CDATA entities in SGML documents is strongly discouraged [12].

While this problem is somewhat mitigated by the more recent inclusion of the `SRC` attribute, there are still problems. This is primarily due to the fact that a document can only be constructed dynamically at the location where a `<SCRIPT>` tag occurs. It's common for this to happen in many places in a document, in which case using the `SRC` attribute results in a network access at each point. That is extremely inefficient, and probably will place more of a burden on the HTTP server then if it had just constructed the document itself.

```
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function compute(form) {
   if (confirm("Are you sure?"))
      form.result.value = eval(form.expr.value)
   else
      alert("Please come back again.")
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter an expression:
<INPUT TYPE="text" NAME="expr" SIZE=15 >
<INPUT TYPE="button" VALUE="Calculate" ONCLICK="compute(this.form)">
<BR>
Result:
<INPUT TYPE="text" NAME="result" SIZE=15 >
<BR>
</FORM>
</BODY>
```

Figure 2.6 Example JavaScript event handling program and document, which evaluates in expression in a form (from the JavaScript Authoring Guide [37])

A final issue is how JavaScript handles events. Adding an attribute for each event class on every tag at which it occurs further undermines the separation between document and script. In addition, it makes it difficult to write generic scripts that watch events occurring within a document (as they need to be directly tied to the document at each place the relevant events might occur).

```
<!ATTLIST FORM
        action  %URI #REQUIRED     -- server-side form handler --
        method  (%HTTP-Method) GET -- see HTTP specification --
        enctype %Content-Type; "application/x-www-form-urlencoded"
        script  %URI #IMPLIED      -- link to client-side script --
        >
```

Figure 2.7 DTD of the Active FORMs tag (from the World Wide Web Consortium's expired HTML 3.0 specification [42])

2.3.3. Active FORMs

As mentioned in Section 2.2.2, the SurfIt! Web browser [2] is a Tcl/Tk-based browser that can execute applets written in that language. In addition to these applets, it supports a type of document scripting called "Active FORMs" [47]. While these scripts are primarily used to specify form interaction, they can also be used in the higher-level context of an entire HTML document.

Unlike JavaScript, the Active FORMs system takes a minimalist approach to the new HTML it introduces. As shown in Figure 2.7, the only markup used is the `SCRIPT` attribute that HTML 3.0 introduced to the `<FORM>` tag. When the SurfIt! browser encounters a form with this attribute, it retrieves the script referenced and begins executing it in a safe Tcl/Tk environment. A script initially executes at the **form** level, allowing it to interact with the user in that way. Once running, however, it can change its execution level so as to manipulate other parts of the HTML document. Active FORMs defines four of these levels, summarized in Table 2.1.

Scripts control the behavior of the browser where they are executing by using the "`applet`" command. This command allows the script to do such things as retrieve documents, append markup to the currently displayed document, access the Tk widgets that make up a form, load data from the network, etc. In addition, the

Level	Description
Browser	The top-level browser application
Hyperwindow	A browser window, which displays Web documents
Hyperpage	A particular HTML document being displayed by a Hyperwindows
Form	The management of an HTML form inside a Hyperpage

Table 2.1 The four possible execution levels of an Active FORMs script [47]

```
proc HMapplet_item {type name value win} {
  switch $type {
    text {
      bind $win <KeyPress> {--procedure that enforces field type--}
    }
    .
    .
    .
  }
}
```

Figure 2.8 Example Active FORMs script that constrains the data in a form field [47]

script can define various "browser call-in" functions, which the SurfIt! browser will call to inform it of events that occur. In particular, the script can be informed when anchors are selected, form elements are created, or a document has finished loading. Note that there are no call-ins for actual user interaction. Instead, an Active FORMs script attaches to the "new form element" call-in, and binds to each newly created Tk widget as is appropriate. An example of this is shown in Figure 2.8. In the example, "`HMapplet_item`" is the browser call-in for a new form widget.

Since the SurfIt! browser is written in the same language as the script it executes, the two tend to be tied closely together. This is a problem, as mentioned where Java was discussed in Section 2.2.1. Also, the Active FORMs system depends heavily on its scripts' ability to manipulate the underlying browser's Tk objects, making it unsuitable as a general-purpose scripting architecture.

2.4. Combining Client-side Services

Most of the recent work in client-side services has focussed on finding ways for the various services described previously to work together. A wide range of approaches have been taken to this problem, ranging from direct combinations of system implementing a single service, to high-level work on the conceptual model and HTML syntax needed. When combining client-side services, the hope is that the resulting combination will exhibit most of the advantages of its individual services, while mitigating their weaknesses.

2.4.1. Eòlas's Web API

In their article "Proposing a Standard Web API" [11], Doyle *et al.* present a system that attempts to combine all three client-side services (plug-ins, applets, and document scripts) by integrating two architectures. The first architecture is a modification of the Eòlas "weblet" that was described in Section 2.1.1, with some modest extensions to help support applets. To that architecture is added the NCSA CCI system described in Section 2.3.1, which provides the weblet plug-ins with some document scripting functionality. The complete system is presented as a suggestion of a "standard API" that can be implemented by browsers. Such a standard would allow plug-ins to be written that work with any browser supporting the described API.

The primary change the Web API makes to the historical "weblet" architecture is the addition of new messages for button and keyboard events, as shown

Message	Description
`DHOEserverUpdate`	Tells a client to update data
`DHOEserverReady`	Tells a client the server is ready
`DHOEserverExit`	Tells a client the server is exiting
`DHOEserverConfigureWin`	Tells a client to resize/reposition the DHOE window
`DHOEclientAreaShown`	Tells the server the DHOE area is exposed
`DHOEclientAreaHidden`	Tells the server the DHOE area is being hidden
`DHOEclientAreaDestroy`	Tells the server the DHOE area is being destroyed
`DHOEbuttonDown`	Sends mouse-pointer coordinates to the server on button down
`DHOEbuttonUp`	Sends mouse-pointer coordinates to the server on button up
`DHOEbuttonMove`	Sends mouse-pointer coordinates to the server on button move
`DHOEkeyDown`	Sends the corresponding *keysym* to the server on key down
`DHOEkeyUp`	Sends the corresponding *keysym* to the server on key down

Table 2.2 Messages between browser (client) and plug-in (server) in Eòlas's Web API [11]

in Table 2.2. The messages form the core of its "distributed hypermedia object embedding" (DHOE) plug-in protocol. Note that, due to the protocol's historical use as a mechanism for embedding large visualization systems into Web browsers, it uses the words "client" and "server" to refer to the browser and plug-in, respectively (contrary to what might be expected). In addition to the new messages, the system adds a mechanism through which X widgets can be embedded in a document, allowing for the creation of more interactive embedded objects than were possible in the original "weblet" architecture. The mechanism was used to create "WebWish," a server that encapsulates a full Tcl/Tk interpreter.

The CCI architecture used in the Web API system is identical to that discussed in Section 2.3.1, with all its related weaknesses. To make use of the CCI interface, a server must connect to the appropriate socket of the browser (though the system does not define explicitly how it should determine which socket to use). Once connected, the server uses the socket-based protocol to execute CCI operations and the X event DHOE protocol for graphical functionality. Such a design allows the system to exploit existing protocols and services, greatly simplifying the API.

There are, however, a number of important limitations to the Web API architecture. The DHOE protocol is little more than a plug-in architecture; its lack of argument passing is a particular limitation in the implementation of true applets. The Web API's use of CCI for document scripts is an even more serious limitation. In addition, the Web API system provides no way to write "pure" document scripts, independent of any embedded applet.

2.4.2. HTML `<OBJECT>` Specification

In the paper "Inserting objects into HTML" [22], Ragget *et al.* propose a new HTML tag named `<OBJECT>`, for embedding arbitrary objects into HTML documents. It is intended to combine the functionality of the `<EMBED>` and `<APPLET>` tags (as discussed in Sections 2.1.2 and 2.2.1) with a uniform HTML syntax.[2] The tag is the first true attempt at combining these two services, and lays a basic foundation for how applet and plug-in extensions will be implemented by browser developers and researchers. Because the proposal has the blessing of the World Wide Web Consortium, it is widely expected to become the standard mechanism for embedding objects into HTML documents.

[2] In addition, the tag includes much of the new image functionality that was first defined in the ill-fated HTML 3.0 `<FIGURE>` tag [42].

The DTD for the `<OBJECT>` tag is shown in Figure 2.9. Like the `<APPLET>` tag, its content can include arbitrary body text and `<PARAM>` tags. Any body text is displayed only if the browser does not support the `<OBJECT>` tag or for some other reason is unable to display the actual object. The `<PARAM>` tags are used to pass arguments to the object, much like the `<APPLET>` tag's similar mechanism. In addition, there are many attributes used to control the rendering and position of the object.

The work-horses of the `<OBJECT>` tag are the four attributes `DATA`, `TYPE`, `CLASSID`, and `CODETYPE`. The first two are used to indicate a *data file* to display, much like the `<EMBED>` tag's `SRC` and `<TYPE>` attributes. The last two, in contrast, specify an actual script that is to be downloaded by the Web browser and executed locally. The scripting attributes are essentially the `<APPLET>` tag's `CODE` attribute, with the addition of a mechanism for specifying the type of script that is to be executed. Finally, the `<APPLET>` tag's `CODEBASE` is used directly, to allow for the specification of a base class path when embedding Java applets.

Two examples of the `<OBJECT>` tag in use are shown in Figure 2.10. The first corresponds to a plug-in, showing how a traditional data type (here, an AVI animation) can be embedded in a document and displayed by an external code module. If the browser is unable to display that particular data type, the HTML markup inside the `<OBJECT>` tag is displayed instead; here, that is a static image. The second example of the tag corresponds to a Java-style applet. It uses the `CLASSID` attribute to indicate the URL of the applet that is to be executed (in this case a Python language script) and supplies one argument to the applet.

While the `<OBJECT>` tag provides a well-defined and flexible syntax for combining plug-in and applet services, it makes no attempt to specify how this integration is to be implemented inside the browser. It is up to each browser developer to

```
<!ELEMENT object - - (param | %bodytext)*>
<!ATTLIST object
  %attrs      -- id, class, style, lang, dir --
  declare (declare) #IMPLIED -- declare but don't instantiate flag --
  classid %URL     #IMPLIED -- identifies an implementation --
  codebase %URL    #IMPLIED -- some systems need an additional URL --
  data    %URL     #IMPLIED -- reference to object's data --
  type    CDATA    #IMPLIED -- Internet media type for data --
  codetype CDATA   #IMPLIED -- Internet media type for code --
  standby CDATA    #IMPLIED -- message to show while loading --
  align   %Align   #IMPLIED -- positioning inside document --
  height  %Length  #IMPLIED -- suggested height --
  width   %Length  #IMPLIED -- suggested width --
  border  %Length  #IMPLIED -- suggested link border width --
  hspace  %Length  #IMPLIED -- suggested horizontal gutter --
  vspace  %Length  #IMPLIED -- suggested vertical gutter --
  usemap  %URL     #IMPLIED -- reference to image map --
  shapes  (shapes) #IMPLIED -- object has shaped hypertext links --
  name    %URL     #IMPLIED -- submit as part of form --
  >

<!ELEMENT param - O EMPTY -- named property value -->
<!ATTLIST param
  name      CDATA             #REQUIRED -- property name --
  value     CDATA             #IMPLIED  -- property value --
  valuetype (DATA|REF|OBJECT) DATA      -- How to interpret value --
  type      CDATA             #IMPLIED  -- Internet media type --
  >
```

Figure 2.9 DTD of the `<OBJECT>` tag [22]

```
<OBJECT data=TheEarth.avi type="application/avi">
  <img src=TheEarth.gif alt="The Earth">
</OBJECT>
```

(a)

```
<OBJECT classid="http://monty.cnri.reston.va.us/grail/
                          demo/clocks/analogclock.py">
  <PARAM NAME=size VALUE=40>
</OBJECT>
```

(b)

Figure 2.10 Example of the `<OBJECT>` tag: (a) embedding a plug-in to display an AVI file; (b) embedding a Python language applet [22]

decide how that is to be done, and exactly how the implementation will make use of the various attributes for defining the data and script. This lack of specification leaves unresolved many of the problems of existing client-side services, particularly the lack of interoperability (both in plug-ins and scripts) between different brands of browsers.

2.4.3. Microsoft's ActiveX

Microsoft has proposed the ActiveX system [31] as a general solution to the problem of integrating client-side services across browsers. ActiveX is built on top of the existing OLE architecture, a set of services for implementing document-centric systems. The system defines various OLE interfaces that encapsulate different parts of the Web architecture. The basic interfaces include such things as how URLs and networking protocols can be represented as OLE objects. In addition, OLE interfaces are defined for scripting languages, objects embedded in HTML documents, document layout, and style control.

```
<OBJECT ID="HelloWorldButton" WIDTH=96 HEIGHT=32
 CLASSID="CLSID:D7053240-CE69-11CD-A777-00DD01143C57">
  <PARAM NAME="Caption" VALUE="HELLO WORLD!">
  <PARAM NAME="Size" VALUE="2540;846">
  <PARAM NAME="FontCharSet" VALUE="0">
  <PARAM NAME="FontPitchAndFamily" VALUE="2">
  <PARAM NAME="ParagraphAlign" VALUE="3">
  <PARAM NAME="FontWeight" VALUE="0">
</OBJECT>
```

Figure 2.11 ActiveX's use of the `<OBJECT>` tag [30]

For its high-level HTML syntax, the system makes use of the `<OBJECT>` tag (discussed in Section 2.4.2) for embedding objects and the `<SCRIPT>` tag (Section 2.3.2) for document scripts. Figure 2.11 shows an example of the `<OBJECT>` tag being used to embed an ActiveX object. The object is indicated with the "`CLSID:`" URL, which provides an OLE object identifier. If the object is not a built-in part of the browser or known to be available on the local system, a URL from which it can be retrieved may be specified with the `CODEBASE` attribute.

The system encapsulates scripting languages as another OLE object, allowing it to support multiple document scripting languages. The standard implementation includes two languages, VBScript and JScript. The former is a lightweight and "safe" version of VisualBasic, while the latter implements the same syntax as Netscape's JavaScript. In addition, an extension to the `<SCRIPT>` tag allows an alternative way of specifying event handlers, as shown in Figure 2.12; the new syntax associates a complete `<SCRIPT>` body with a particular event.

ActiveX also supports Java applets, although they exist as a somewhat distinct entity. The Java interpreter is itself encapsulated in an OLE object: however, it does not support the scripting interface. Such a design means that Java programs

```
<form name="Form1">
        <input type="button" name="Button1" value="Click">
        <script for="Button1" event="onClick" language="VBScript">
                alert "Button has been pressed"
                document.Form1.Button1.value="PRESSED"
        </script>
</form>
```

Figure 2.12 ActiveX's extension to the `<SCRIPT>` tag [30]

can not be used as document scripts, even if the program is retrieved through the `<SCRIPT>`'s `SRC` attribute[3]. Java applets are also embedded into documents through the traditional `<APPLET>` tag, rather than being integrated with the rest of the `<OBJECT>` syntax.

While Microsoft claims its ActiveX objects are "language-neutral" and promotes them for use as traditional applets [29], such a claim is not entirely accurate. ActiveX objects are actually low-level 80x86 machine code (hence "language-neutral," only in the sense that any language can be compiled to machine code) retrieved from the network and executed directly on the client machine. Executing code like this is an important security concern, as there is no ability to protect code at the machine instruction level. Once an ActiveX applet is retrieved and execution begins, it has complete access to the local system. In fact, there are already ActiveX "applets" that do such things as turn off the host computer. Microsoft has tried to address the security issue by incorporating authentication into ActiveX objects. While authentication is helpful for large-scale applications, it is question-

[3]With the `<SCRIPT>` tag Java would be difficult to support as a document scripting language in any case, since Java programs are stored as raw binary files

able whether it is a solution for the more traditional domain that applets are used in—that is, small pieces of customized code that are integrated seamlessly into Web pages.

In addition, the ActiveX architecture is dependent, to a large degree, on the underlying OLE system in Windows. In an environment that requires data to be usable on as wide a variety of platforms as possible, it is doubtful that such a platform-specific system will be useful as a true general solution. Microsoft has announced plans to port OLE and ActiveX to other platforms, but much of the power of ActiveX lies in the wide number of existing OLE objects that can be used with it. Almost all these objects are dependent on the complete underlying Windows operating system, leaving the utility of a higher-level architecture port questionable.

2.4.4. Netscape's LiveScript

Netscape has introduced "LiveScript" [38] as a system whose primary goal is the integration of the browser's three client-side services: Netscape plug-ins, Java applets, and JavaScript document scripts. The integration is accomplished by creating specific channels of communication between the different services, allowing one to control the operation of another. By creating channels, one service gains indirect access to the capabilities of another service, with the LiveScript architecture serving as intermediary.

Figure 2.13 shows a high-level view of the architecture. There are four distinct communication paths defined: (1) plug-ins can call Java methods; (2) Java can call native methods implemented in a plug-in; (3) JavaScript can call Java methods; and (4) Java can call methods implemented in JavaScript. Note that there are no

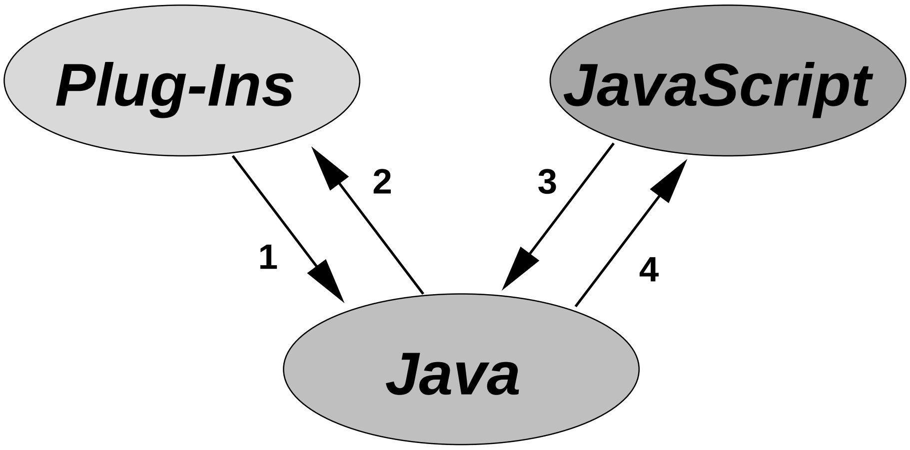


Figure 2.13 Netscape's LiveScript architecture [38]

direct communication channels between plug-ins and JavaScript. Such channels can, however, be implemented in two hops, by going through the Java interpreter.

To call Java methods from within a plug-in, Netscape supplies a program that converts a Java class definition into C preprocessor macros providing access to the class. For example, the `substring` method of the String class is called with the macro `java_lang_String_substring`. Netscape's Java Runtime Interface [36] is used as the underlying mechanism for connecting the two. It also provides Java with access to methods defined by the plug-in.

Within JavaScript, the Java class hierarchy is made available through the `Packages` object. For example, JavaScript can access Java's System class with the code `var System = java.lang.System`. A message can then be sent to the Java console by executing the JavaScript code `System.err.println("Greetings from JavaScript")`. Similarly, the applets in a document can be accessed through JavaScript's document object, which exposes them in its `applets` array. For example, an applet whose `NAME` attribute is "`myApplet`" can be accessed as `document.applets["myApplet"]`. In the other direction, Netscape exposes its JavaScript environment through the `netscape.javascript.JSObject` object. For example, a handle on the document's window can be retrieved with `JSObject.getWindow(this)`.

While LiveScript provides a mechanism for communication between the client-side services included in Netscape, it does little to unify their architectures. For example, there is still no way to actually write an applet with JavaScript. The `JSObject` does provide a method to evaluate actual JavaScript code, but at best this is a awkward way to write a JavaScript program. Similarly, having to call Java methods in plug-ins as a means to access the browser services needed to implement new applet languages leaves a lot to be desired, and there is still no way to write a plug-in that adds a new document scripting language.

2.5. Collecting Document Usage Statistics

As the Web grows to assume an increasingly important role in business and publishing, it will be more and more necessary to collect information about who is using a content provider's Web pages, and how they are doing so. Such information traditionally has been very difficult to discover, due to the Web's distributed nature and the characteristics of its underlying protocols. There are, however, a number of systems actively being designed to help in this area. In addition, the computer-based nature of the Web presents many new opportunities for content providers to examine the ways in which the information they supply is being used (or, perhaps, not being used). By applying traditional human-computer interaction research to the use of Web pages, it may be possible to discover an unprecedented amount of information that is extremely difficult to collect from traditionally published material.

2.5.1. HTTP Hit Statistics

The original, and still most widely used, source of Web usage statistics is the document "hit count." This is a simple count of the number of times a particular document has been retrieved. Each time an HTTP server [4] receives a request for one of its documents, it writes a line of information about the request in an access log. As shown in Figure 2.14, the line includes the machine that requested the document, user identification and authentication, date and time, the HTTP method (including the path to the document), the server status code, and the number of bytes transferred. The first line of Figure 2.14, for example, shows a request that came from the machine "ada.cs.orst.edu" with no user identity or authentication, on Nov. 20, 1995 at 11:53:05 PDT, and was a GET method on the file "/~hackbod/ihtml/", with the server successfully fulfilling the request and sending back 1720 bytes of data.

To turn a low-level access log into useful information, a program must examine the data and compile it into higher-level statistics. One of the most common operations performed is to count the number of times each page occurs in the log, resulting in the familiar hit count. There are, however, other useful data that can be extracted from the log, such as the number of hits originating from different domains

```
ada.cs.orst.edu - - [20/Nov/1995:11:53:05 -0800] "GET /~hackbod/ihtml/ HTTP/1.0" 200 1720
...orst.edu - - [20/Nov/1995:11:53:15 -0800] "GET /~hackbod/ihtml/environ_ex.html HTTP/1.0" 200 2447
ada.cs.orst.edu - - [20/Nov/1995:11:53:21 -0800] "GET /~hackbod/ihtml/sc95poster/ HTTP/1.0" 200 3522
...- - [20/Nov/1995:11:53:40 -0800] "GET /~hackbod/ihtml/sc95poster/architec.html HTTP/1.0" 200 4526
... - - [20/Nov/1995:11:53:40 -0800] "GET /~hackbod/ihtml/sc95poster/architec.gif HTTP/1.0" 200 5506
ada.cs.orst.edu - - [20/Nov/1995:12:18:33 -0800] "GET /~hackbod/ HTTP/1.0" 200 2
```

Figure 2.14 Example HTTP access log produced by NCSA's "httpd" server and machines, number of hits during different time periods, number of requests for non-existent documents, total number of bytes transferred, etc.

Unfortunately, HTTP access logs have many limitations, largely due to the stateless nature of the underlying HTTP protocol. The basic problem is that there is no concept of a session—every request for a document is processed without any context information that can connect it with the requests that arrive before or after it. In theory, the user identity and authentication fields could be used to associate lines in the access log; however, the former is rarely sent to the browser (as shown in the example) due to privacy concerns, and the latter is not useful in the most common case, where no password is required to access the document. All this means that there is no way to track inter-document access paths through the server's documents (e.g., identifying that a user first viewed `document1`, then `document2`, then `new/document5`).

Another problem is that even the basic hit statistics may not be extremely accurate. Caching—both by the browser and in proxies between the browser and server—greatly reduces the number of requests for a document that the server actually sees. Conversely, if a user asks the browser to reload the document, hits are recorded that do not reflect additional users viewing the document. Access logs provide no information of finer granularity than the level of an entire document. Thus, there is no way to determine how long a document was viewed, nor other interesting information about how a particular document is being used.

2.5.2. Gathering Consumer Statistics

Connolly [9] summarizes recent proposals for extending various Web protocols to provide inter-document usage statistics and other marketing-related information. He then discusses the pros and cons of each proposal, examining the issues that are involved in its implementation and the types of information that can be collected. Of particular importance to this discussion are the privacy issues involved in collecting such information.

Three protocols are discussed in the paper. The first proposes that a "`Request-ID`" HTTP header, be sent from the browser to the server. It would be composed of two fields, a random session number (generated by the browser when it started executing), followed by a request count (incremented each time the browser sends an HTTP request). This header would supply all information needed by an HTTP server to track user sessions across documents.

The second protocol employs profiles as a simple way for servers to collect demographic information from forms submitted by browser users. A number of special field names (`profile-full-name`, `profile-postal-street`, `profile-email-address`, etc.) are defined for use in hidden form fields. When a browser implementing the protocol submits a form, it fills in each field given one of these special names with information previously saved by the user as a set of user profile preferences—allowing the information to be collected automatically without the user having to fill in form fields each time they are needed.

The third protocol is intended to help HTTP servers track per-session state information. It makes use of the existing Web authentication protocol, defining a new "anon" scheme. When an HTTP server receives a request, it sends back a reply requesting authorization for "anon" authentication. Upon receiving this response, the browser sends a new request with authorization for the "anon" user, whose password is a unique 128-bit random number identifying the browser.

After describing these protocols, Connolly then discusses four issues that they (or similar usage collection protocols) should address:

1. There is a conflict between the desire to collect usage information and the privacy rights of consumers. The article states that the `Request-ID` protocol seems to address these concerns, by allowing browser users to control the information being sent to the server. In addition, various "norms" of behavior must be assumed, such as the service provider keeping the collected data private.

2. There is a need to create Web services involving persistent state. Many applications implemented on Web servers need to keep track of information about a user across individual HTTP requests (for example, to keep track of a user's shopping cart while browsing an on-line store). This is addressed by the anonymous authentication protocol.

3. The desire to collect demographic information unobtrusively also conflicts with the privacy rights of consumers. Content providers have a need to collect information about their consumers without imposing an undue burden on them, yet consumers have the right to control what information is collected. This is addressed by the profile-based protocol, in two ways. Not only is it the user who must enter the information into the browser in the first place, but it is assumed that the browser displays all the information it will be sending the the server when a form is submitted.

4. The need to cache documents affects all three proposed protocols. It is imperative that any statistics-gathering protocol consume as little Internet bandwidth as possible, which means they shouldn't break the cacheing of documents unless that is strictly necessary to implement a desired feature. Further, there needs to be some way for cacheing proxies to collect the demographic infor-

mation included in a form so that it can be relayed back to the originating server.

2.5.3. HTTP Cookies

Kristol and Montulli's [24] proposes a system for creating "stateful sessions with HTTP requests and responses." This allows HTTP servers to track individual browsers as their users move through the server's documents. While its primary purpose is to help servers associate state information with browser sessions (for such things as electronic shopping carts), the system also provides a mechanism for tracking inter-document usage statistics.

The system centers around two new HTTP headers, `Set-Cookie` and `Cookie`. A session is initiated by the server sending a document with the `Set-Cookie` header, the syntax for which is shown in Figure 2.15. The header is composed of a semicolon-separated list of argument/value pairs. The first argument is an arbitrary cookie name and value to set and is followed by a number of optional arguments to control the cookie's behavior. When a browser receives the `Set-Cookie` header, it stores the cookie data, then sends back a similar `Cookie` header when making future requests for documents from the server. The `Cookie` header includes the name and value pair(s) originally sent by the server, thus creating a mechanism for identifying sessions unambiguously and allowing the server to store state information on the client machine.

To use this system for tracking document usage, a server can simply generate a unique cookie (by incrementing a global counter, for example) whenever it receives an HTTP request with no cookie information. Browsers supporting this protocol will then send the cookie counter back to the server when requesting future documents.

```
set-cookie = "Set-Cookie:" cookies
cookies    = 1#cookie
cookie     = NAME "=" VALUE * (";" cookie-av)
NAME       = attr
VALUE      = value
cookie-av  = "Comment" "=" value | "Domain" "=" value
           | "Max-Age" "=" value | "Path" "=" value
           | "Secure"            | "Version" "=" 1*DIGIT
```

Figure 2.15 Syntax of the Set-Cookie HTTP header.

By storing an HTTP request's cookie value in the server's access log, it is then a simple matter for programs processing the log to follow session trails through it.

2.5.4. Intra-document Usage Statistics

None of the Web document usage statistics collection methods discussed so far have addressed intra-document usage statistics—that is, how individual documents are being used. For example, knowing that there has been a hit on a certain document says nothing about how useful it was. Yet there are many usage details that a content provider may wish to know. Did the browser user find it easy to locate information in the document? Are users making use of the information in it? If there are forms in the document, are they well-organized? How long does a user typically spend looking at the document? Does the user read through it sequentially, or jump around to different locations? Collecting intra-document usage information to help answer the above questions enters the realm of traditional software usability research. Because of the distributed and dynamic nature of the Web, collecting such data should be unobtrusive and not require that the user be located in a traditional usability lab.

One technique that might be useful for collecting intra-document usage information is software logging. For example, Neal and Simons describe a system called "playback," which records the keypresses of a user interacting with a system, allowing the entire session to be re-played and examined in detail at a future time by the experimenter [34]. The system consists of an external CPU that is placed between the user's keyboard and the machine running the software to be tested. The external CPU records every key pressed by the user, tags it with a time stamp, and sends it on to the machine under test. Once the user has finished a session, the experimenter can review the recorded keystrokes and play them back through the tested machine to watch the user's progress. More importantly, the recorded session can be played back at an accelerated rate, typically allowing an eight-hour session to reviewed in around two hours. In addition, the system supports the testing of documentation by allowing an experimenter to view the user's activity through a video monitor and insert comments about the user's actions into the electronic record. Neal reports that the system has been used successfully in an iterative product evaluation of IBM BASIC, as well as for testing the usability of form interfaces, panel navigation, and multiple windows [35].

Morris *et al.* report on an extension to the playback system that integrates video and audio records with the keyboard information [32]. During replay, the video and audio recordings of the user are synchronized with the keyboard information, allowing the researcher to observe not only what the user entered into the system and how the system responded, but also the user's physical and verbal activity. The system can record multiple input channels, allowing the recording of additional things such as mouse input. While they report that the system has been of value in getting feedback on interface performance, the researchers comment on two important limitations of playback. Reviewing and analyzing the collected data is a labor intensive task, for which they suggest the development of tools that will automate the process. In addition, when evaluating more sophisticated software it can be difficult to get it to respond to recorded input in the same way it did for the user. Their system attempts to address this second problem by recording the software's output as well as user input, but further work needs to be done in the area of graphical displays.

One method of automating the task of analyzing data logs is reported by Siochi and Ehrich, in which a user session transcript is analyzed for repetitions of actions [44]. They postulate that repeated patterns in a transcript may provide valuable information about the user's interaction with the system. A repeated sequence may indicate such things as commands being at too low a level for the desired tasks, the user circumventing an application malfunction, the user searching for a way to do something, a malfunction causing loss of state information, etc. The method revolves around their Maximal Repeating Patterns (MRPs) algorithm, which searches a text transcript for repeated patterns that are "as long as possible, or an independently occurring substring of a longer pattern." To analyze a transcript, it is first transformed into a normalized list of commands executed by the user. The MRP tool examines the list for repeating patterns. The result of this analysis not a metric of usability, but rather an indication of what parts of the transcript might be "interesting" to the experimenter.

Siochi and Ehrich include describe experience of using the MRP method to test GIPSY, an image processing system. They tested the system with 17 users, collecting a total of 17,086 command lines for MRP analysis that resulted in 3,523 detected MRPs. From this data, 12 classes of problems were discovered, of which only four had been previously known. The experimenters also found the MRPs useful for preparing and conducting future interviews with two of the users and discovered that presenting the users with the raw transcripts of the points of interest in their sessions was significant in helping the users recall the task they were performing. However, the method is limited in terms of the information that can be collected, that information being specific, detailed problems rather than general aspects of a system's usability. In addition, the method is specifically aimed at command line interfaces, ignoring the more common direct manipulation interface.

Smith *et al.* report on a more sophisticated method of collecting and analyzing session log data [45]. Their system makes use of a higher-level data representation, the action protocol. Rather than specific user input, an action protocol records general data such as movement between windows, selection of objects, etc. This allows the system to analyze log data that already has been interpreted by the software under test, lessening the problems of interpreting the meaning of that data during transcript analysis. Each action results in a line of information in the transaction session, containing such information as the name of the action, start and stop times, and other relevant parameters. Since the software under test was specifically customized to process this high-level session log, it can also replay the recorded session exactly as it appeared to the user.

For analysis, a session transaction is viewed "as analogous to a statement or discourse in a natural language." A grammar can then be constructed to represent the language with the actions recorded during the session as terminal symbols; nonterminal symbols are "higher level cognitive behavior." Smith *et al.* describe three grammars they have written for these sessions, representing models of the user's cognitive processes while working with the system. To help analyze and interpret the data, they developed a number of tools for visualizing and animating both the collected transcripts and how the user interacts with the system.

Currently, usage information about hypermedia environments has only been collected using dedicated systems. Trumbull *et al.* present one example of collecting this kind of information, analyzing a program they developed called Bughouse, which presents information about cultural entomology [48]. Users can access information in the system in three different modes. Index mode presents a familiar book-style cross-index, Browse mode allows the user to move through the information using the metaphor of a Victorian country house and its grounds, and Guide mode tracks the user's progress and uses that information to suggest to the user other areas for investigation. As a user works with the system, it records a log of time spent in individual areas of the program, the search mode used, and activity within each area. After having the users fill out questionnaires about their reactions to the system, the researchers examined the session logs. From these, they classified users into four categories, based on the types of search modes used, and found marked differences in the amount of information found by the users. In particular, users who primarily used the browse mode tended to locate more of the relevant information and be fairly sure they had found everything they needed. Index users, in contrast, were confident they'd found everything, but had actually discovered much less than the browse users.

Software logging cannot, of course, take the place of traditional HCI evaluations. In particular, it provides no explanation about why the user may be performing a particular action [50]. While the collected data can be augmented with explanatory comments, video and audio records, and user questionnaires [16], the need to collect additional information defeats many of the advantages of unobtrusive and easy collection of usage information. Further, the data collection process itself always raises important ethical issues about whether or not users are informed that their actions are being recorded.

3. OVERVIEW OF INTERACTIVE HTML

The iHTML system combines the three most commonly used client-side services—plug-ins, applets, and document scripts—into one uniform, flexible architecture. This is accomplished by defining a client-side service that is a superset of the three services, supporting all the capabilities they offer. An important goal of iHTML was to design a system that is both browser- and language-neutral. The latter was accomplished by simply combining the client-side services, placing all code for implementing applet and document scripting languages in the external plug-in. Since the system defines the interface between the external scripting code and the browser functionality needed to implemented applets and document scripts, all that is needed to add a new language is to write write a plug-in that interfaces it with the rest of the iHTML system. The goal of browser independence was addressed by creating a well-defined interface between the client-side services and the browser, so that iHTML does not depend on any of the browser's underlying implementation details.

This chapter first presents a discussion of the new concepts and protocols that the iHTML system introduces to traditional Web browsers. After that, a high-level overview of its design is presented, explaining the different parts of the system and the division of functionality between them. Finally, the issues of security and protection are examined and the approach iHTML takes to addressing them is described.

3.1. iHTML Concepts

The iHTML system introduces a number of new concepts and protocols to the traditional Web browser's architecture. The most fundamental are the ways that files of different types are handled, and the generalization of applets and JavaScript-style programs into one basic client-side scripting mechanism. To support these concepts, iHTML implements and introduces some extensions to the World Wide Web Consortium's `<OBJECT>` tag. [22] Also, iHTML introduces a new URL scheme for integrating low-level network socket operations with the rest of the browser's URL system.

3.1.1. File and Script Types

Under iHTML, all types of files that may be embedded within an HTML document (e.g., images, applets, animations, etc.) are handled using one uniform mechanism. This is accomplished by extending the traditional "plug-in" concept to handle client-side scripts associated with documents. These new plug-ins are called "language modules" and have additional access to the browser so that they can implement the types of functionality expected of applet and JavaScript-style programs.

A file's type is identified through its MIME "`Content-type`" header. When the iHTML system prepares to display a file, it uses the MIME content type information to associate it with any available language module that knows how to "display" (or execute) the file. The language module is dynamically loaded by the library, and completely encapsulates the code needed to display a particular type of file.

For file types that correspond to executable programs, the iHTML system distinguishes between two particular classes of program. The first class is the iHTML **applet**, which operates almost identically to the familiar Java applet. An applet is a program that appears embedded within an HTML document and interacts with the user primarily through the user interface it displays there. The second program class is the **document script**, which does not have such a close analogy to any existing client-side scripting technique. Conceptually, it is similar to a JavaScript-style program, in that it is associated with an entire document and executes in that context. However, a document script uses a fairly different syntax and conceptual model; this will be discussed more in Chapter 4.

An important point is that both types of scripts use the same underlying architecture. They are implemented by the same language module, and (as is reasonable) they have the same access to the browser's functionality. The only difference is the context—window-specific or whole-document—in which the script executes.

3.1.2. HTML Syntax

To support the unification of plug-ins, applets, and document scripts, the iHTML system makes use of the `<OBJECT>` tag. The tag was designed to provide a uniform interface for embedding any type of object into a document (here, plug-ins and applets), and works well as intended. iHTML implements a subset of the tag, shown in Figure 3.1, that provides enough information for the system to find an appropriate language module to display a file's data, along with the information needed by applets.

Figure 3.2 shows an example of the tag being used to embed an iHTML applet. The example also illustrates the unification of plug-ins and applets. When the tag is encountered, the system will first retrieve "`myanim.mpg`" and try to find a language module to play the MPEG animation. If such a language module does not exist, iHTML will fall back to retrieving the "`plaympeg.py`" script, handing the MPEG data to it to be displayed.

```
<!ENTITY % Context "(document|module|applet)">

<!ELEMENT object - - (param | %bodytext)*>
<!ATTLIST object
  %attrs                       -- id, class, style, lang, dir --
  classid  %URL     #IMPLIED   -- identifies an implementation --
  data     %URL     #IMPLIED   -- reference to object's data --
  type     CDATA    #IMPLIED   -- Internet media type for data --
  codetype CDATA    #IMPLIED   -- Internet media type for code --
  context  %Context #IMPLIED   -- context object executes in --
  height   %Length  #IMPLIED   -- suggested height --
  width    %Length  #IMPLIED   -- suggested width --
  name     %URL     #IMPLIED   -- submit as part of form --
  >

<!ELEMENT param - O EMPTY -- named property value -->
<!ATTLIST param
  name     CDATA     #REQUIRED  -- property name --
  value    CDATA     #IMPLIED   -- property value --
>
```

Figure 3.1 DTD of iHTML's implementation of the World Wide Web Consortium's <OBJECT> tag [22]

```
<OBJECT data="myanim.mpg" type="video/mpeg"
        classid="plaympeg.py"
        classtype="application/x-ihtml; language=python">
```

Figure 3.2 Example markup to embed an iHTML applet

```
<OBJECT context=document classid="watch.py"
            classtype="application/x-ihtml; language=python">
```

Figure 3.3 Example markup to create an iHTML document script

The `<OBJECT>` tag was not, however, originally created to support document scripts. To apply it to that use, iHTML extends the tag, allowing it to appear in an HTML document's `<HEAD>` section and adding a new attribute named `CONTEXT` (see Figure 3.1). When iHTML encounters the tag in a document's header and finds its `CONTEXT` attribute to "`DOCUMENT`," the system executes the associated script in the context of the entire document.[1] Figure 3.3 shows an example of this use. Here, the system retrieves the "`watch.py`" program as before, but the program has access to document-level events and services.

3.1.3. HTML Parse Trees

In order to support many of the operations expected of document scripts—particularly the ability to dynamically create document text—the iHTML system provides an abstract interface to the underlying browser's HTML parse tree representation. Every HTML document corresponds to a well-defined tree structure. As an example, consider the document shown in Figure 3.4. When parsed into its internal representation, this document becomes a tree of nodes, where `<HTML>` is

---

[1]Since a script executing in a document context is not provided with an area on the HTML document in which to display itself, the behavior of file types (such as images) that need a display area is not defined when used this way. Currently, they are simply ignored.

```
<HTML>
  <HEAD>
    <TITLE>Example Document</TITLE>
  </HEAD>
  <BODY>
    <H1>Example Title</H1>
    <P>Example paragraph.</P>
    <HR>
    <ADDRESS>Example address</ADDRESS>
  </BODY>
</HTML>
```

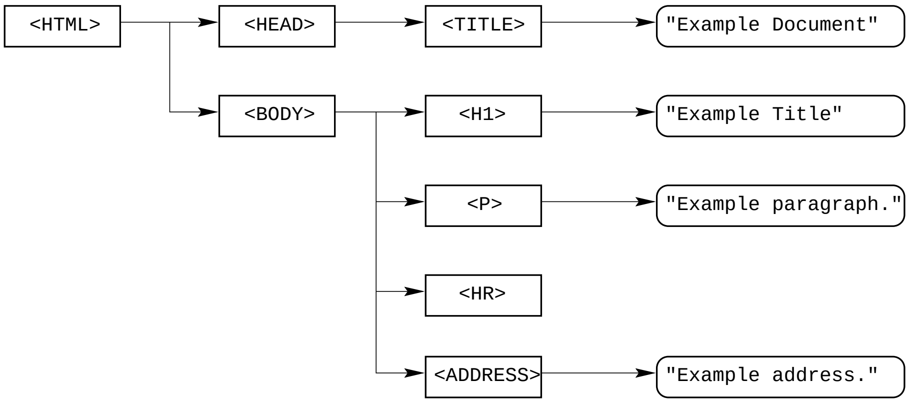
Figure 3.4 A simple HTML document

the root of the tree. A standard representation of such a document is shown in Figure 3.5, as well as a more traditional top-down tree view.

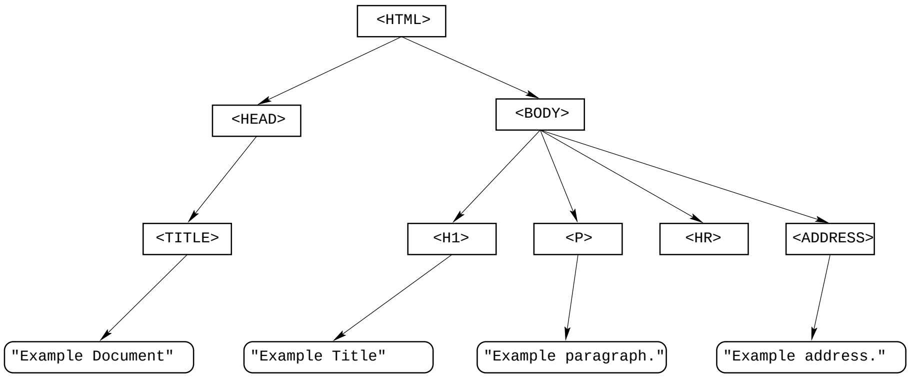
The HTML tree structure, including standard operations to create and modify it, is an integral part of the system. It provides a well-defined representation of the document, allowing a document script or other programs executing in the system not only to construct a document dynamically by creating new nodes, but also to go back and examine or modify existing HTML documents.

3.1.4. URL Extensions

Finally, iHTML defines a new URL scheme named `tcp:` that creates a low-level interactive transaction, encapsulating the basic BSD `socket()` mechanism into the same interface as URLs. The complete syntax of the `tcp:` scheme is "`tcp://`*`machine`*`:`*`port`*`/`", where *machine* is the DNS name of the remote machine with which to make the socket connection and *port* is the port address of that socket. Once a URL with this scheme is opened, data from the remote machine appears as



(a)



(b)

Figure 3.5 Parse tree of the document in figure 3.4: (a) The parse tree representation of the document; (a) A more traditional tree view

it becomes available, as for other URLs. In addition, data can be sent back to the remote machine, allowing it to support interactive sessions.

An actual `tcp:` URL can take one of three distinct forms, controlling its basic behavior:

`tcp://`*`machine`*`:`*`port`*`/` The fully-qualified form creates a socket connection with a computer named *machine* at the given *port*. This corresponds directly to a `socket()` call.

`tcp:`*`port`*`/` The same as the fully qualified form shown above, but connects with that machine from which the document was retrieved.

`tcp://`*`port`*`/` Listen for socket connections coming in to the given port. This corresponds to a `listen()` call.

3.2. General Architecture

The iHTML system is divided into three major components, as shown in Figure 3.6. The **iHTML Library** is the central regulator of the system, which controls the interaction between a Web browser and the available back-end language modules. It is the only component that is actually implemented in the system distribution. For other components, only the interface is defined; their implementations are left to the discretion of the component implementors. The arrows in the figure represent the possible ways that control flow can move through the system; for example, the arrow between the Language modules and browser component indicates that language modules can directly access browser services, while no arrow in the other direction shows that the browser does not directly manipulate the language modules. The browser component regulates all interaction with the user, while the iHTML library component provides the public interface to the back-end language modules and is responsible for loading and unloading them as needed.

A **Language Module** is the implementation of one or more MIME content types, and tells the rest of the system how to display or otherwise handle data of

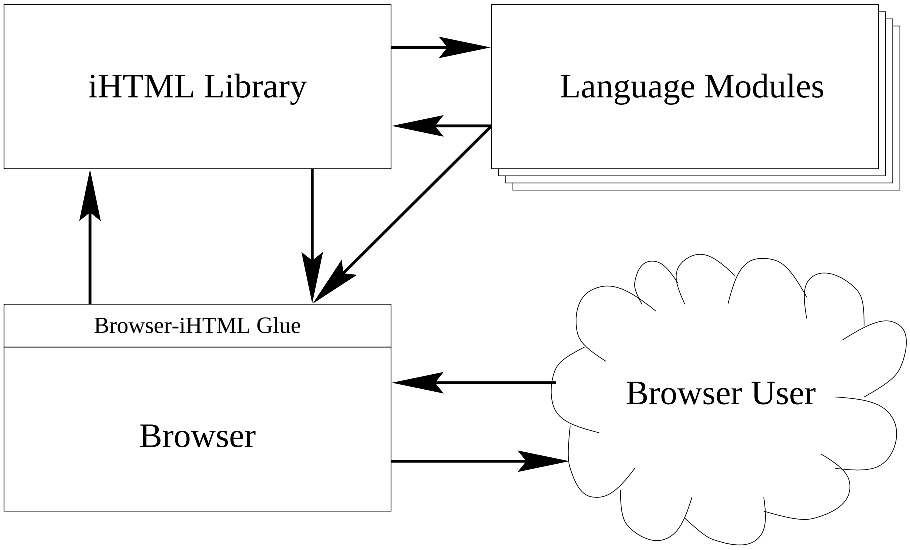


Figure 3.6 High-level architecture of the iHTML system, showing the three main components and communication between them

that type. There can be many different language modules available, each handling a different type of data. The system dynamically loads these modules as needed. A language module is very similar to a traditional Netscape-style plug-in, except that its interface to the rest of the browser has been extended to support various types of client-side scripting.

The **Browser** component is the traditional Web browser. The iHTML architecture defines a standard interface between it and the rest of the system, but otherwise treats it as a black box. Thus, the "Browser-iHTML Glue" division represents the code used to map between this black-box implementation and the interface defined by iHTML. The glue layer includes both the functions that must be supplied by the browser component and the calls it must make into the iHTML Library component in order for the system to operate correctly.

3.2.1. Browser Services

The browser is the main component and driver of the system. It is responsible for most interaction with the user, for parsing and displaying HTML documents, parsing and retrieving URLs, and managing the display and resources. This functionality is exposed to the rest of the system through a set of functions and types defined by the iHTML architecture. Any functionality in the browser not specifically defined by this architecture is a black box and may be implemented however the browser chooses.

In this high-level view, the browser component also encapsulates some functionality that is more specifically a part of the operating system, such as managing graphical objects, drawing, and dynamically loading shared libraries. While the iHTML architecture does not make any direct distinction between the two components, the interface glue to a particular operating system is typically implemented separately from the browser-specific pieces, so that it does not need to be re-written for each browser. The services provided by the browser component fall into nine broad categories

The browser must supply a set of functions that provide access to, and manipulation of, its internal **HTML parse tree** representation. These routines work on an opaque data type, the HTMLNode. All operations—including those to traverse the tree, get information about a node, delete and insert a node, create a new node, and access the browser's HTML parser to create a whole tree—are performed through function calls, allowing the browser to use any underlying data representation it would like. The representation must provide a true tree structure, allow access to each node's type, attributes, and the text associated with it, and be able to associate two arbitrary C pointers (the "language" and its "object" that owns the node) with any node.

For **documents and browsers**, the library defines the opaque type IH-Browser to represent a browser window, and the type IHDocument to represent a handle on a single HTML document (currently displayed in that window or part of its history stack). The handles are used by functions to associate the iHTML Library's resources with a document, to retrieve and set the root HTMLNode of the HTML tree being displayed in a document, and to inform the browser of changes to the currently displayed tree.

When **retrieving URLs**, the opaque type IHRetrieveRequestP represents an active URL-based network transaction. The browser must supply functions for creating a request from a URL string, accessing data as it becomes available from the network, writing data to the transaction, and terminating the request.

The iHTML **event** system is built on top of the underlying event system provided by the operating system. It defines the IHEvent type, a structure that contains information about where the event occurred, its basic class and code, and any object associated with it. The browser supplies functions for allocating and duplicating events, getting and setting attribute values associated with an event (such as the mouse position, key pressed, etc.), and deallocating events. This allows it to append its own data after the publicly defined structure, for storing internal state information.

There are a number of opaque types defined by the library for implementing graphical **user interfaces**, including IHWidgetRep (the browser-side representation of a user interface element), IHPixmap (an off-screen display buffer), IHColor (a rendering pen), IHFont (a text rendering font), and IHFutureCall (a function call that is to occur at a later time). The user interface types—and associated browser functions for allocating, manipulating, and destroying them—allow the system to create and interact with user interface widget hierarchies. While the functions are typically implemented as part of the operating system glue, they may be implemented by the browser itself if it needs more control over its display.

Typically implemented by the operating system part of the browser glue, the architecture defines routines a browser must supply for loading, examining, and unloading **shared libraries**. They use an opaque symbol, IHSharedModule, as a handle on the shared library.

In addition to the major groups of functions already described, there are many **miscellaneous** functions that do not fit into a particular group. These are used for such things as getting version information or information about the capabilities of the browser (for example, whether it supports tables), showing status information about the progress of an operation or a hyperlink, parsing directory paths, and creating temporary files.

3.2.2. iHTML Services

The iHTML Library is the only component that is directly implemented by the iHTMLsystem. It defines all the non-opaque types used in the system, along with a set of functions used to integrate the various components with each other. The library essentially serves as the traffic cop for the client-side scripting services. It manages the language modules in the system and receives documents and events submitted by the browser. The iHTML Library is responsible for determining what to do with them and ensuring that they are handled by the correct language module.

In addition, the iHTML Library is the only component that is granted direct interaction with a language module. Because other components also need to interact with the language modules, the library must supply a number of functions that duplicate the functionality of the language services, but provide them as a language-independent interface. When called, the functions direct actions to the appropriate language module. Library services fall into five broad categories.

As part of its **document interface**, the library supplies functions that the browser calls to submit a document to the system, detach a document from the system, and start or stop the scripts associated with a document. Submitting a document to the system results in the allocation of a IHGlobalInfo type, which contains the state information that the iHTML Library must keep with a document; all further interaction with the document occurs through this type. A number of additional types may be connected to this handle, as the system runs: the IHModuleInfo contains information about an associated document script, and one IHEmbedded-Info type is allocated for each applet embedded in the document.

In its role of managing **language modules**, the iHTML Library defines the type IHLanguage to expose information about a single language module, and provides functions for loading and releasing language modules by name. In addition, it supplies general functions for accessing the HTML and Widget objects that are created by the back-end language modules.

The library defines a **data buffer**, a piece of memory containing arbitrary data that can be manipulated by removing and appending data to it. This is used, for example, by the browser URL services as a place to collect data as it becomes available over the network. Functions allocate, manipulate, and deallocate these buffers, which are represented by the type IHBufferP.

There are many places where the library must manage arbitrary lists of resources, so it defines a standard set of functions for creating and using **doubly**

linked lists. The linked list functions operate on the IHList and IHNode types. Operations are available for performing all the familiar list operations on them.

The library also includes a number of **miscellaneous** functions that don't fit into any particular group. Examples include the functions for initializing and terminating the entire system and for retrieving version information about the library, as well as a function the browser calls to dispatch an event (causing it to be routed to the correct language module).

3.2.3. Language Services

Language modules represent the back-end functionality of the system, supplying the code needed to handle particular data types. They exist as separate, dynamically loaded libraries, loaded and initialized as needed. Their interface is intended to be accessed only by the iHTML Library itself. Four general categories of services are provided.

A number of functions are supplied for managing the language module's internal representation of an **HTML parse tree**. They include functions for determining the range of HTMLNodes contained in a tree and managing its reference count (so that the language module can deallocate it when it is no longer part of a document or a larger tree).

The system defines a common type, IHLangEnv, used to hold state information about a **language environment** executing on a particular document. Functions are included for creating and deleting language environments, as well as starting and stopping the programs in an environment. In addition to the common parts of the structure defined by the library—including handles on the iHTML Library's

IHGlobalInfo and the language's IHLanguage with which it is associated—the language can append any private state information it needs.

To manage **scripts**, functions are supplied for initializing and deleting the IHModuleInfo data for a document script associated with an entire module, and the IHEmbeddedInfo data associated with each applet. Functions are also available to start and stop individual applets. Each language module defines an opaque data type IHCodeInfo, which contains its internal representation of a script program (e.g., the compiled form of the script) and can be attached to its associated IHModuleInfo or IHEmbeddedInfo.

Similar to the browser's HTMLNode, IHWidgetRep also has a language-side object associated with it that is used to manage the **user interface**. The language module must manage this object and be able to attach it to the browser-side widget with which it is associated. In addition, the module provides a function that is called by the system to dispatch user interface events to the module.

3.3. Security and Protection Issues

Whenever dealing with client-side services in Web browsers, the issues of security and protection must be addressed. Downloading and executing code without the user's intervention raises some obvious security issues that can't be ignored. However, since the underlying mechanism for implementing the scripting languages is based on low-level language modules that have been compiled to machine code, this is an issue that the system can not enforce directly. Instead, the system provides a number of services that attempt to help help the language developer enforce these security and protection issues when writing a language module.

In general, the iHTML system and its language modules rely entirely on the underlying security features of the network and operating system for authentication and other security services. This typically means that, unless other measures are taken, scripts can have potentially full access to a particular user's account or even an entire machine.

In network scripting, this security problem is typically mitigated by providing some kind of **restricted execution** environment in which a script can run. iHTML attempts to support this design by distinguishing two modes for script execution, **unlimited** and **protected**.

Scripts that have been retrieved from a the file system of the local machine (or, in languages where it is appropriate, that have been retrieved from particular directories of the file system) operate in unlimited mode. In this mode, a script is granted as much access to the underlying machine as its language provides. Unlimited execution typically means that, at the very least, a script can access the file system directly, open arbitrary sockets, and has unrestricted access to information belonging to the user.

Any other script—particularly those retrieved over the network—should be assumed by a language module to be malicious, and placed under protected execution. Such scripts are prevented from accessing any "dangerous" functionality, including the file system or other low-level access to the operating system that the language may have. To help support this protection, the iHTML system only allows such scripts to open `tcp:` URLs that connect with the same machine from which they were retrieved; other, dangerous URL schemes (such as `file:`) are disabled.

As previously stated, it is up to each language module to enforce these protections. As an example, the Python module does so by allowing scripts executing in protected mode only to access functions defined by the basic iHTML system described in the previous section, plus certain very select parts of the standard Python environment (such as its math module).

4. SCRIPT DEVELOPER INTERFACE

From the Interactive HTML script developer's point of view, iHTML supports two distinct varieties of scripts. These are called "applet-level" and "document-level," corresponding to the traditional forms of HotJava applets (Section 2.2.1) and JavaScript programs (Section 2.3.2). Under iHTML, the only important difference between the two types of scripts is how they are linked with, and interface with, an HTML document. Beyond that, both scripts have the same basic functionality and work in the same way.

This chapter describes how a program written by a script developer accesses the iHTML library's functionality, using the Python language module as an example of a typical object-oriented interface to iHTML. The reader is assumed to have basic familiarity with the Python language [27], including an understanding of Python modules, types, and classes. The language module itself represents a fairly straightforward encapsulation of the iHTML system. Most concepts are directly translated to the higher-level Python language, exposed as a set of basic Python types and classes. Other scripting language interfaces will usually present a similar interface to the library, expressed in the concepts and syntax appropriate to the particular language.

4.1. Applet-level Scripts

An iHTML applet-level script is a program that executes in the context of a graphical area embedded within a larger document. It does not normally have direct access to that document, but is only concerned with the user events and graphics that occur within its area. An applet-level script is associated with a document by inserting an `<OBJECT>` tag in the body of the document. (See Section 3.1.2 for details on the tag.) A Python applet script is retrieved from the network as Python source code.[1] Once the code has been retrieved, it is compiled and then executed by the Python interpreter in the context of a new Python module. The resulting module's namespace must contain within it a callable object named `ihEmbed`, which is the entry point for the applet.

4.1.1. The Application Class

The standard iHTML module **ihApp** defines the Python language module's high-level applet interface, and includes a class named Application that encapsulates its functionality. An active applet is represented by an instance of the Application class (or subclass), which must be returned by the `ihEmbed` symbol when it is called. Thus `ihEmbed` typically is defined as a subclass of Application, since Python's semantics for calling a class is to instantiate it and return the new instance to the caller.

The parameter to `ihEmbed` is a dictionary of keyword arguments, corresponding to the `<PARAM>` tags included with the applet's `<OBJECT>` tag. In addition, the following standard symbols are included in the argument dictionary:

`Width` is the value supplied to the `<OBJECT>` tag's `WIDTH` attribute.

`Height` is the value supplied to the `<OBJECT>` tag's `HEIGHT` attribute.

`_Document_` is the document the applet is located in. The Document class is described in Section 4.2.1.

---

[1] Python programs must be transmitted over the network as source code because the format of Python's byte-code changes between versions of the interpreter.

The Application class's constructor expects to be given this dictionary of arguments when it is instantiated, so that it can handle some of the common low-level details involved in processing them. In particular, it understands how to interpret the `Width` and `Height` parameters, using the requested size to allocate space for the applet in the document.

4.1.2. Program Structure

Figure 4.1 show a "minimalist" applet-level script. It first imports the **ihApp** module (in line 2) to access the Application class. The next program statement (line 6) declares the applet's entry point `ihEmbed` to be a subclass of Application. Lines 9 through 12 define the subclass's constructor—it simply passes the applet's keyword arguments up to the Application superclass. Finally, line 15 exposes the module's `ihEmbed` symbol, making it publicly accessible. The `__export__` symbol used here is not a standard part of the Python language itself, but an extension that the Python language module adds to implement its security restrictions. All the program in Figure 4.1 does is reserve display space in the document. To the user, it appears as an empty rectangle that doesn't respond to any user events.

4.1.3. Graphics Rendering and Widgets

To actually do something, an applet must make use of the **ihWidget** module. The module defines two new classes (Container and Widget) and three new types (Pixmap, Color, and FutureCall). Together, they provide the core functionality needed to organize the applet's display, draw graphics, and respond to user actions.

The Container class, summarized in Table 4.1, is the main organizational and functional component of the module. A container defines a rectangular screen

```
# Get standard application class definitions.
import ihApp

# Our own application is a subclass of the iHTML-defined
# Application class.
class ihEmbed(ihApp.Application):

    # Class initialization.
    def __init__(self,**args):

        # Just send control on up to the superclass.
        apply(ihApp.Application.__init__,(self,),args)

# Make our program externally visible.
__export__ = ['ihEmbed']
```

Figure 4.1 A minimal applet-level script written in Python

area in which it is positioned. Every Container object appears inside a parent Container object (or the top-level Application object), and can itself have one or more embedded child Containers. This organization allows the applet's display area to be subdivided into a hierarchy of functional components, each managed by its own Container object.

A hierarchy of containers knows how to position itself and all its subcomponents. The Container class has methods that its parent class can use to query its preferred, maximum, and minimum dimensions. Thus, a Container can examine its children to determine their dimensions and use that information to compute its own dimensions and position the children appropriately within its larger display area. In addition, the Container class has a basic set of methods that the script developer needs to draw graphical shapes (such as lines, rectangles and arcs) on the display, to draw text, and to copy images from one area to another.

Class Methods	
Method	Description
`Background`	Set or get the current screen background color, the color that is initially filled into an area of the container that is newly exposed
`BackPen`	Set or get the current drawing background color
`Destroy`	Remove the container from the applet
`Dimensions`	Get the container's position and size
`DoResize`	Change the container's physical size and location
`DrawArc`	Draw an unfilled arc in the current foreground color
`DrawLine`	Draw a line in the current foreground color
`DrawRectangle`	Draw a rectangle in the current foreground color
`DrawText`	Draw a line of text in the current foreground color
`FillArc`	Draw a filled arc in the current foreground color
`FillRectangle`	Draw a filled rectangle in the current foreground color
`Flush`	Force all drawing operations to complete
`FutureCall`	Execute a function call at a future time
`ForePen`	Set or get the current drawing foreground pen
`KeyState`	Get the current pressed/released state of the modifier keys
`Layout`	Size and position the Container's children
`MakeColor`	Allocate a new palette color
`MakePixmap`	Create a new off-screen rendering area
`MaxDimensions`	Compute the maximum size of the Container
`MinDimensions`	Compute the minimum size of the Container
`MousePosition`	Get the current mouse (x,y) position
`MouseState`	Get the current pressed/released state of the mouse buttons
`OnRedraw`	Method called when the Container needs to redraw itself
`OnRedraw`	Method called when the Container needs to resize itself
`Paste`	Draw a Pixmap into the Container
`PrefDimensions`	Compute the Preferred size of the Container
`TextExtent`	Measure the dimensions of a line of text, if rendered in the current font

Class Members	
Member	Description
`_Document_`	*Read-only.* The Document the Container is on
`Children`	*Read-only.* An array of the Container objects that are direct children of this one
`Height`	*Read-only.* The current height, in pixels, of the Container
`Width`	*Read-only.* The current width, in pixels, of the Container
`X`	*Read-only.* The current left edge of the Container within its parent
`XOffset`	*Read-only.* The current left edge of the Container relative to its closest parent Widget
`Y`	*Read-only.* The current top edge of the Container within its parent.
`YOffset`	*Read-only.* The current top edge of the Container relative to its closest parent Widget

Table 4.1 Summary of the Container class's public interface

The final responsibility of the Container is managing other user interface resources, which are encapsulated in the three basic types defined by the **ihWidget** module. The Pixmap is an off-screen buffer that can be drawn into with the same rendering methods provided by the Container. The Color type represents a single colored pen that is used for drawing. The third type, FutureCall, encapsulates a function call that is to occur at some later date, supporting the implementation of timed or repetitive execution (required for such things as animation). The Container class provides three methods used to create objects of each type.

The Widget class, a subclass of Container, is summarized in Table 4.2. It expands on its parent class in two ways. The most obvious of these is event handling. User actions that occur within a Widget's display area are collected by that Widget and dispatched by it to its event handlers. In addition, the Widget creates its own **display context**. This has some important implications with regard to the original Container class's functionality. The most important is that the coordinate system in a Widget is offset such that the point (0,0) is always at the top left corner of the Widget, and all rendering is clipped to the boundaries of the Widget. A plain Container, in contrast, shares the display context of its closest parent widget, meaning the script developer must be more careful to manually position and clip any drawing being done in a container.

Finally, to connect the preceding discussion with the topic of writing applets, the Application class (which encapsulates an active applet) is a subclass of the Widget class. This means that an applet has access to all the functionality of its Widget and Container superclasses, including the ability to receive user interface events, create and manage other Container and Widget objects within its display area, and draw directly into the Application display area itself.

Class Methods	
Method	Description
`OnEnter`	Method called when Widget receives an `Enter` event
`OnEvent`	Method called when Widget receives an event that does not otherwise have an event-handler defined
`OnKey`	Method called when Widget receives a `KeyPress` or `KeyRelease` event that is not handled by one of the next two methods
`OnKeyPress`	Method called when Widget receives a `KeyPress` event
`OnKeyRelease`	Method called when Widget receives a `KeyRelease` event
`OnLeave`	Method called when Widget receives an `Leave` event
`OnMouse`	Method called when Widget receives a `MouseMove`, `MousePress` or `MouseRelease` event that is not handled by one of the next three methods
`OnMouseMove`	Method called when Widget receives a `MouseMove` event
`OnMousePress`	Method called when Widget receives a `MousePress` event
`OnMouseRelease`	Method called when Widget receives a `MouseRelease` event

Class Members	
Member	Description
`XOffset`	*(Defined in superclass.)* This member is always set to zero for instances of a Widget
`YOffset`	*(Defined in superclass.)* This member is always set to zero for instances of a Widget

Table 4.2 Summary of the Widget class's public interface

4.2. Document-level Scripts

The iHTML document-level script is a program that executes as part of a complete HTML document. It has a global view of the document, and thus is able to examine and manipulate the objects in the document, as well as the document itself, and to react to user-interface events occurring in the document. A document-level script is associated with a document by inserting an `<OBJECT>` tag in the head of the document and setting the tag's `CONTEXT` attribute to "`DOCUMENT`". (See Section 3.1.2 for details on the tag.) The script is retrieved from the network and compiled just as described in Section 4.1 except that, instead of `ihEmbed`, the script's main entry point is now the symbol `ihMain`.

4.2.1. The Document Class

The **ihDoc** module defines a class named Document that is very similar to the applet script's Application class. The class encapsulates an active document script's context and provides the script with an interface to the document itself. Summarized in Table 4.3, the Document class allows the program to examine the capabilities of the system it is executing under, add and remove event handlers for the entire document, and display basic status information. In addition, it provides access to the HTML parse tree representing the document. Section 4.2.3 will describe how this parse tree can be examined and modified.

Information about the capabilities of the browser and operating system in which a program is running is available through the Document's `Capabilities` member. The member's attributes are summarized in Table 4.4. A program can use this information to customize a document's presentation for the machine it is running under, by doing things such as changing the images it uses and replacing unavailable capabilities (such as tables) with some alternative representation.

Events are distributed by the Document class to a list of handler functions. The handler list is managed with the `AddHandler` and `RemHandler` methods. The handler functions in this list are called for each event that reaches the Document object, in the order they were added to the list. When called, a handler function is handed one argument, representing the event that occurred. iHTML events are discussed in further detail in Section 4.3.1.

Class Methods	
Method	Description
`AddHandler`	Add an event handler to the document
`Reformat`	Make the browser re-examine and display the current document
`RemHandler`	Remove an event handler from the document.
`ShowLink`	Show user information about the destination of a hyper-link
`ShowProgress`	Display a textual progress report to the user

Class Members	
Member	Description
`Capabilities`	*Read-only.* A dictionary describing the capabilities of the browser and host operating system that the program is running under
`DocHeight`	*Read-only.* The total height of the document, in pixels
`DocWidth`	*Read-only.* The total width of the document, in pixels
`ViewBottom`	The bottom edge of the user's view within the entire document, in pixels. Set to move the view
`ViewHeight`	*Read-only.* The amount of the document vertically visible to the user at any one time, in pixels
`ViewLeft`	The left edge of the user's view within the entire document, in pixels. Set to move the view
`ViewRight`	The right edge of the user's view within the entire document, in pixels. Set to move the view
`ViewTop`	The top edge of the user's view within the entire document, in pixels. Set to move the view
`ViewWidth`	*Read-only.* The amount of the document horizontally visible to the user at any one time, in pixels
`VisHTMLTree`	The current HTML parse tree being displayed as the document

Table 4.3 Summary of the Document class's public interface

4.2.2. Program Structure

Figure 4.2 shows a "minimalist" document-level script, which is very similar to the applet script presented in Section 4.1.2. It first imports the **ihDoc** module to access the Document class (line 2), then declares its entry point `ihMain` to be a subclass of Document. Lines 9 through 15 define the subclass's constructor; the constructor here passes the applet's keyword arguments up to the Document super-

Capability	Description
forms	Indicates if the browser supports forms
tables	Indicates if the browser supports tables
sockets	Indicates if socket connections are supported/allowed
comp_mosaic	Indicates if the browser supports the standard X-Mosaic capability set
disp_text	Indicates if the browser supports a textual interface
disp_graphics	Indicates if the browser supports a graphical interface
disp_mono	Indicates if the browser supports monochrome graphics
disp_color	Indicates if the browser supports palette-based color
disp_true	Indicates if the browser supports true-color

Table 4.4 Browser capability information available

```
 1. # Get standard document class definitions.
 2. import ihDoc
 3.
 4. # Our own document is a subclass of the iHTML-defined
 5. # Document class.
 6. class ihMain(ihDoc.Document):
 7.
 8.     # Class initialization.
 9.     def __init__(self,**args):
10.
11.         # Just send control on up to the superclass.
12.         apply(ihDoc.Document.__init__,(self,),args)
13.
14.         # Display the document.
15.         self.Reformat()
16.
17. # Make our program externally visible.
18. __export__ = ['ihMain']
```

Figure 4.2 A minimal document-level script written in Python

class, then allows the HTML document that is executing in to be displayed. Line 18 exposes the module's `ihMain` symbol to public access.

To the user, a document associated with the script in Figure 4.2 behaves no differently than it would if it had no document script. The script simply tells the browser to display the HTML as it normally would, then ignores all user interaction.

4.2.3. Manipulating HTML Markup

One of the most important capabilities of a document-level script is that of examining and modifying the actual document the user is viewing. As was discussed in Section 3.1.3, the HTML parse tree representationis an integral part of the iHTML system. The **ihMarkup** module provides access to the browser-side HTML parse tree, allowing a script to construct HTML documents on the fly, change the document in response to the user's interaction with it, and extract data from a document with which it is associated. The module provides an interface through two basic types, HTMLTree and HTMLCursor. New objects of these types can be created with the functions `NewHTMLTree()` and `NewHTMLCursor()`, respectively, in the **ihMarkup** module.

The HTMLTree object represents an actual HTML tree or sub-tree (i.e. it is composed of one or more connected HTML nodes). It doesn't provide any access to that tree, however. Instead, an HTMLCursor object is used to reference a particular node on an HTMLTree. It is through the cursor that the HTMLTree and its associated nodes may actually be examined and manipulated. The HTMLCursor type is summarized in Table 4.5. It provides three major classes of functionality: movement through a tree of HTML nodes, examination of the current node, and insertion of new nodes into a tree.

In addition to the standard types of movement in a tree—go to child, parent, next, or previous node—the cursor also defines methods that move to nodes with

Type Methods	
Method	Description
MoveChild	Go to the child of the current node
MoveNext	Go to the next node in this level of the HTML tree
MoveNextDepth	Go to the next node of the tree, in depth-first order
MoveNextID	Go to the next node, in depth-first order, whose ID attribute matches a specific value
MoveNextTag	Go to the next node, in depth-first order, that is a specific tag type (e.g., the next <P> or <H2> node)
MoveChild	Go to the parent of the current node
MovePrev	Go to the previous node in this level of the HTML tree
MovePrevDepth	Go to the previous node of the tree, in depth-first order
SetPos	Change the cursor's location—to the start of an HTMLTree, to the same location as another HTMLCursor, or so that it is not on any node
PasteAfter	Insert a HTML tree after the node the cursor is located at
PasteBefore	Insert a HTML tree before the node the cursor is located at
PasteHead	Insert a HTML tree as the first child of the node the cursor is located at
PasteTail	Insert a HTML tree as the last child of the node the cursor is located at

Type Members	
Member	Description
CurHTMLTree	*Read-only.* The HTMLTree that the cursor is currently located on
First	*Read-only.* True if the cursor is located on the first node of a level
Last	*Read-only.* True if the cursor is located on the last node of a level
Leaf	*Read-only.* True if the cursor is located on a node at the bottom of a tree
NodeAttr	*Read-only.* Contains the attributes of the current node
NodeText	*Read-only.* If the current node is a text node, the text it contains
Root	*Read-only.* True if the cursor is located on a node at the top of a tree
Tag	*Read-only.* The tag type of this node (e.g., <H1>, <HEAD>)

Table 4.5 Summary of the HTMLCursor type's public interface

particular types and `ID` attributes, or move through a complete tree in depth-first order. Depth-first is the order that nodes appear in an actual HTML document, so this feature makes it easy for a program to perform searches through an entire document. The HTMLCursor also provides various types of information about the "current" node. The `Tag` member identifies the type of node it is (e.g., a `<HEAD>`, `<TABLE>`, or a text node). For nodes representing actual HTML tags in the document, the `NodeAttr` member can be used to access the attribute values associated with that tag (for example the `HREF` attribute of an `A` tag would be `HTMLCursor.NodeAttr.HREF`). Otherwise, `NodeText` contains the raw textual information associated with a comment or text node.

Finally, a cursor can be used to actually change the tree it points to, by applying various "paste" methods. These take another HTMLTree as an argument, and serve to insert that tree into a location in the HTMLCursor's tree, near the node pointed to by the cursor. Note that performing an insert does not copy the tree, but instead embeds it inside the cursor's tree. This means that some of the nodes between the trees are connected to each other, so moving between them may cause the cursor to end up on another tree. The HTMLCursor keeps track of such a possibility, and moves itself between trees as appropriate.

To summarize, Figure 4.3 shows a Python program that dynamically constructs the HTML parse tree previously shown (Figure 3.5). The first two lines start the parse tree, with its outer `<HTML>` node, and place a cursor on it. Lines 3 to 5 insert the `<HEAD>` node inside it, move to that node, and insert the `<TITLE>` sub-tree inside it. Line 6 places the `<BODY>` tag after the header, and line 7 positions the cursor on it. Lines 8 to 11 finally insert the rest of the nodes, repeatedly placing them as the "last" child of the `<BODY>` node. Note that this document could have been

```
tree = NewHTMLTree("<HTML></HTML>")
cur = NewHTMLCursor(tree)
cur.PasteHead(NewHTMLTree("<HEAD></HEAD>"))
cur.MoveChild()
cur.PasteHead(NewHTMLTree("<TITLE>Example Document</TITLE>"))
cur.PasteAfter(NewHTMLTree("<BODY></BODY>"))
cur.MoveNext()
cur.PasteTail(NewHTMLTree("<H1>Example Title</H1>"))
cur.PasteTail(NewHTMLTree("<P>Example paragraph.</P>"))
cur.PasteTail(NewHTMLTree("<HR>"))
cur.PasteTail(NewHTMLTree("<ADDRESS>Example address</ADDRESS>"))
```

Figure 4.3 Dynamic construction of the HTML parse tree shown in Figure 3.5

constructed simply by calling `NewHTMLTree()` with the complete ASCII markup of the document to be constructed.

4.3. Common Scripting Services

The Python language module includes a number of additional components that expose the rest of the underlying iHTML system, which are of mutual interest to both applet and document script developers. These components include modules for retrieving URLs through the underlying browser, basic iHTML event services, and miscellaneous modules for examining information about the environment in which a script is executing.

4.3.1. Retrieving URLs

The **ihURL** module provides basic services for retrieving URLs through its host browser. Most of its functionality is encapsulated in the RawURL class, which

Class Methods	
Method	Description
`Destroy`	Kill the URL transaction
`GetHeader`	Extract URL meta-information, e.g. the header fields of MIME-based protocols
`GetStatus`	Get the current status of the URL connection
`GetURL`	Get a string representing the fully-expanded name of the URL
`OnError`	Handler called when an error occurs
`OnRead`	Handler called when some data from the URL arrives
`Write`	Send data back through the URL

Table 4.6 Summary of the RawURL class's public interface

is summarized in Table 4.6. The constructor to the class requires the Document instance in which the program is executing, plus a string representing the name of the URL transaction, in order to begin. A URL name can be either fully qualified, or relative to the given document; it will be handed off to the underlying browser for processing, with data asynchronously appearing at the new instance's `OnRead()` method as it becomes available. The class also includes methods for writing data to the URL and retrieving status information about the transaction.

The RawURL class can be used with most of the familiar URL schemes, including "`http:`", "`ftp:`", and iHTML's "`tcp:`". However, it restricts access to some insecure schemes. In particular, the "`file:`" URL is not available because it provides direct access to the user's file system.

Finally, the **ihURL** module defines one other class, LineURL, which is a subclass of RawURL. The LineURL class organizes data arriving at the low-level `OnRead()` method into a series of text lines, and sends them to its own `OnLine()` method as they are constructed. This makes it easy to write such things as clients and servers that communicate using lines of commands.

4.3.2. Events

Asynchronous actions that occur within a browser—such as user interface actions or higher-level constructs built on them—are characterized by iHTML as a set of "events" that travel through the system, carrying the action data to its different components. The **ihEvent** module provides access to the iHTML event system, encapsulating it in one new type called the Event. Summarized in Table 4.7, the type provides no methods, but only a set of standard members that are associated with the underlying iHTML event; changing these attribute values changes the data in the event. Every event has four read-only attributes that can be set only when the event is created. `Class` and `Code` together describe the event type, while `Document` and `Object` indicate where it came from.

In addition to the attributes shown in Table 4.7, the Event type allows any other customized attribute to be assigned a value and later retrieved. This is useful when new types of events are being used. For example, a text control that produces an "update" event might include its text in a customized attribute named "text." Note, however, that customized attributes are not a part of the underlying iHTML event system, so they will be lost if the event moves between languages modules.

New Event objects are created by calling the module's `NewEvent()` function. The function must be called with the new event's class and code types, the Document (where the program is executing), and an object that created the event (which may be set to `None`). The class controls the major structure of an event. For example `Document` events make use of the various "`View...`" attributes, while `Widget` events all use the `MouseX`, `MouseY`, `KeyASCII`, etc. attributes. The standard class and code types are listed in Table 4.8.

Type Members	
Member	Description
`Class`	*Read-only.* The identifier for the vent's major class type. See Table 4.8 for the standard types
`Code`	*Read-only.* The identifier for the vent's minor code type. See Table 4.8 for the standard types
`Document`	*Read-only.* The Document on which the event occurred
`KeyASCII`	An integer representing the ASCII keycode associated with keyboard-related events (`KeyPress` and `KeyRelease`)
`KeyState`	The up/down state of the key modifiers (shift, control, caps-lock, alt) at the time the event occurred
`MouseButton`	The mouse button that caused a button-related event (`MousePress` and `MouseRelease`)
`MouseState`	The up/down state of the mouse buttons (select, popup, option) at the time the event occurred
`MouseX`	The current X location of the pointer at the time the event occurred
`MouseY`	The current Y location of the pointer at the time the event occurred
`Object`	*Read-only.* The object that produced the event; usually a Widget instance
`RedrawHeight`	The height of an area associated with the event (damaged area for `Redraw` events, or new object size for `Resize` events)
`RedrawWidth`	The width of an area associated with the event (damaged area for `Redraw` events, or new object size for `Resize` events)
`RedrawX`	The left edge of an area associated with the event (damaged area for `Redraw` events, or new object position for `Resize` events)
`RedrawY`	The top edge of an area associated with the event (damaged area for `Redraw` events, or new object position for `Resize` events)

`ViewBottom`
`ViewFullHeight`
`ViewFullWidth`
`ViewHeight`
`ViewLeft`
`ViewRight`
`ViewTop`
`ViewWidth`
	The position and dimensions (in pixels) of a smaller view on a larger canvas, for `Document` events; `ViewFullHeight` and `ViewFullWidth` are the dimensions of the larger canvas/document and the rest are the position and size of the smaller view

Table 4.7 Summary of the Event type's public interface.

Event Classes	
Class	Description
`Document`	Event occurred in the top-level Document
`Widget`	Event occurred in a Widget

Event Codes	
Code	Description
`Enter`	Pointer entered a user interface area
`KeyPress`	The user has pressed a key on the keyboard
`KeyRelease`	The user has released a ket on the keyboard
`Leave`	Pointer left a user interface area
`MouseMove`	Mouse pointer position has changed
`MousePress`	The user has pressed a mouse button
`MouseRelease`	The user has released a mouse button
`Resize`	A user interface area has changed size
`Redraw`	A user interface area needs to be redraw
`Scroll`	A view into a larger area has scrolled or otherwise been changed

Table 4.8 Standard event class and code types.

4.3.3. Environment Information

The last two components of the iHTML Python language module supply information about the browser, operating system, and language module to the executing script. The first two are contained in the **ihBrowser** module, while the last is in the **ihPython** module. Interfaces to these modules are summarized in Table 4.9. Together, the modules provide detailed information about the machine and user id under which the browser is executing, the name and version of the browser, the version of the iHTML library and when it was compiled, and the version and compile time of the Python language module.

ihBrowser Module	
Symbol	Description
`HostName`	The full name of the local machine the browser is running on
`UserName`	The name of the user running the browser
`BRFullName`	A string uniquely identifying the brand and version of the browser
`BRName`	A string identifying the brand of the browser
`BRRevision`	An integer representing the minor revision code of the browser
`BRVersion`	An integer representing the major version code of the browser
`BuildDate`	a string representing the time and date that this iHTML library was compiled
`IHRevision`	An integer representing the minor revision code of the iHTML library
`IHSubBuild`	An integer representing the unique build code of the iHTML library
`IHVersion`	An integer representing the major version code of the iHTML library

ihPython Module	
Symbol	Description
`BuildDate`	a string representing the time and date that this Python language module was compiled
`Revision`	An integer representing the minor revision code of the Python language module
`SubBuild`	An integer representing the unique build code of the Python language module
`Version`	An integer representing the major version code of the Python language module

Table 4.9 Python environment information available through the **ihBrowser** and **ihPython** modules.

5. COLLECTING USAGE STATISTICS

To present an example of iHTML in action, a system implemented with iHTML's Python language module is examined. The system is used for collecting and displaying intra-document Web usage statistics. In other words, it is designed to watch a user's interaction *within* an HTML document (for example, scrolling through a document's content, printing, or selecting hyperlinks). Such information is not available through previous methods of collecting usage information (such as hit counts), which only measure usage across documents. After briefly describing the server used to track and distribute the usage information, this chapter examines the system's two iHTML Python scripts in detail. The first is a document script that watches the user interact with a page and sends that information back to the server, while the second dynamically displays the information collected by the server.

Figure 5.1 shows the high-level architecture of the usage statistics collection programs. The central component is the **usage server**, a program running on the same machine as the HTTP server from which the HTML document of interest was retrieved. It accepts remote connections from other programs, which can add new information and examine current information. The two remote programs are the

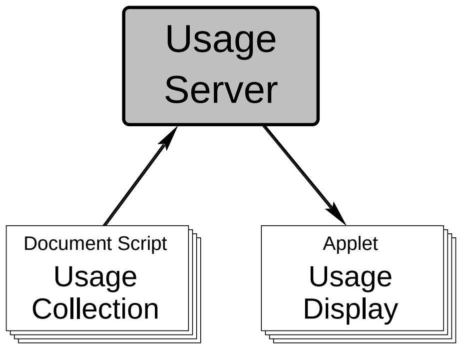


Figure 5.1 Architecture of the usage statistics collection programs.

Character	Line type
*	A **command** to be executed (e.g., "`* close`")
!	An **error** report (e.g., "`! bad mode`"). The network connection may be closed after a line of this type is sent
>	A line of **usage information** collected by the server
:	A **comment** on status (e.g., "`: Connect okay`")

Table 5.1 Initial characters indicating the type of a line when communicating with the usage server.

usage collection and **usage display** clients. The server must be able to accept an arbitrary number of clients at any time, as more than one person may be reading the HTML page being monitored.

The usage server has a simple protocol for communicating with the remote scripts. It is a full-duplex line-based protocol, where data is sent across the network in the form of single lines of text. The first character of a line indicates the type of data. Four types are defined, as shown in Table 5.1. Commands (i.e., "`*`" lines) usually are sent from client to server, to control the server's operation and report on user interaction with the page. Usage information (">" lines) is always sent from the server to a client that is displaying the usage information. The other two types of lines may be sent in either direction and supply additional information on how the various programs are interacting.

The usage server can operate in one of two modes with each remote client. In **send mode**, the client is viewing the document for which usage information is being collected and sending new information to the server. **Watch mode**, on the other hand, is for clients that are displaying the data collected by the server. When a client first connects to the server, it must immediately send a command to indicate the mode in which it will be operating ("`* send`" or "`* watch`", respectively). The client

Command	Description
`* anchor`	Send information about the user moving over or selecting a hyperlink in the document
`* close`	Close connection with server. This is the only command that may also be sent from the server to the client
`* initpos`	Supply the initial position of the user's view on the document
`* print`	Inform the server that the user has printed a hard-copy of the document
`* scroll`	After the view on the document has changed, supply the amount of time the user spent in the *previous* view
`* send`	At start of session, put server into **send mode**
`* watch`	At start of session, put server into **watch mode**

Table 5.2 Commands recognized by the usage server.

Usage Line	Description
`> begin view`	Begin transmitting usage statistics on viewing the document
`> total` *`time`*	The total time all users have viewed the document, where *time* is a decimal number representing seconds
`> div` *`time`*	The total time all users have viewed one piece of the document, where *time* is a decimal number representing seconds
`> end view`	Done transmitting usage statistics on viewing the document

Table 5.3 Usage information sequence sent by the usage server.

may then send additional commands to control the server or supply information about user interaction; these commands are summarized in Table 5.2.

Every time the page server receives new page interaction information from one of its "`send`" clients, it sends a new summary of new usage information to all clients that are registered to watch. The information is transmitted as a series of **usage information** lines, in the order shown in Table 5.3. Each is sent once, except for "`> div`", where a line is sent for each vertical region of the document.

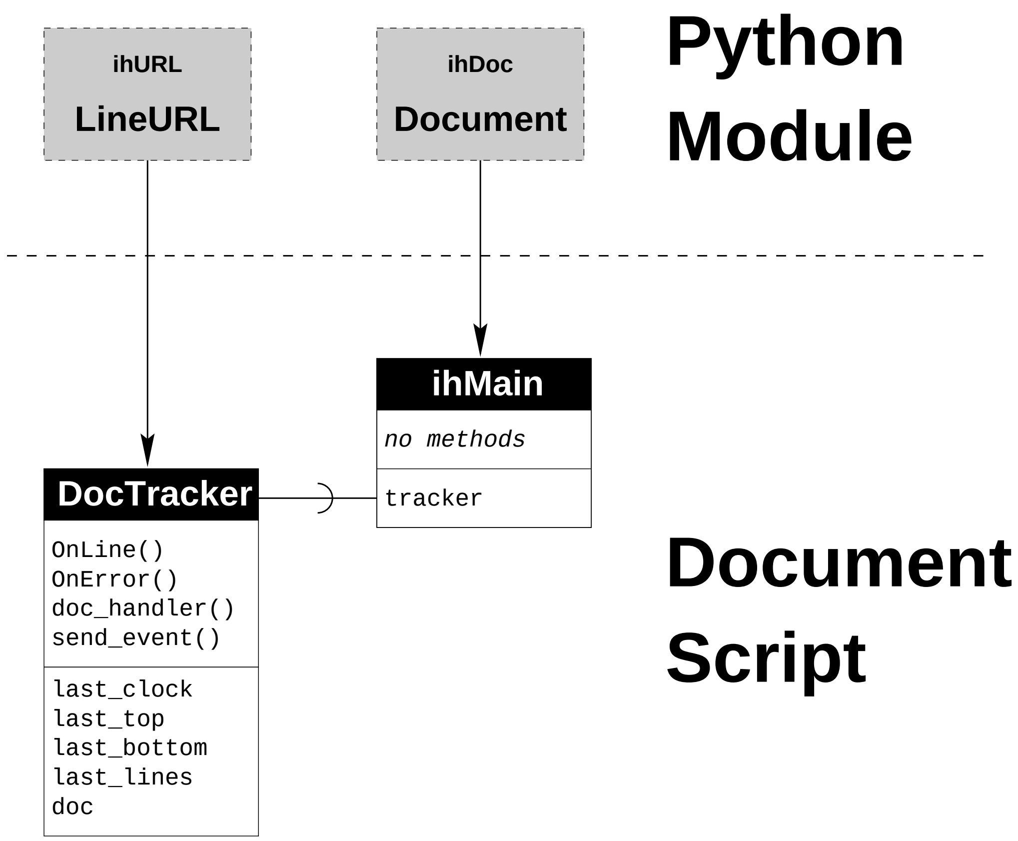


Figure 5.2 Architecture of the usage retrieval program.

5.1. Retrieving User Interaction

The Python usage retrieval program is used as a document script, executing in the document for which usage information is being collected. It consists of two classes, DocTracker and ihMain, as shown in Figure 5.2. The former manages the connection with the usage server, transforming events that occur in the document into commands to the server. The latter is the main entry point of the program, which parses the program arguments, creates the DocTracker class, and attaches that class's event handler to the document.

The program first initializes some variables for the default host and port that it will connect with (if they are not supplied in the document script's argument list). In addition, it retrieves the various Python and iHTML modules it will use during its execution.

Next comes the definition of DocTracker class, a subclass of the Python language module's LineURL class used to create remote URL transactions that exchange data as series of newline-terminated strings of text. The DocTracker class's initialization method requires two arguments, supplying the document object where it is running and the URL to open. As its last action, it sends a "`* send`" command to the usage server, informing the server that the session is to be one in which new user interaction data is sent to the server. The LineURL class defines two event handling methods, `OnLine()` and `OnError()`. The former is called when one or more lines of input are ready; the method used here simply echoes the lines to the browser as progress reports. The latter is used to report error conditions, and also results in progress reports being sent to the browser.

The method `doc_handler()` handles an actual iHTML event, extracting the event code and handing it off to another method, `send_event()`, to be processed. The `send_event()` method's parameters are the name of a user interface action to send to the usage server, and an optional event object with additional information. It examines this information, and sends the appropriate command. If the action is view-related ("`initpos`" or "`scroll`"), the method sends to the server the information about the *last* view the user had on the document, stores the current view position, and resets the view timer.

The second class defined is ihMain, the main entry point of the document script. Its initialization method determines the URL for the usage server. If the argument "`Server`" is supplied, that argument's value is used; otherwise, it constructs its own `tcp:` URL from the default host and port. A DocTracker then is created from that URL, the browser's progress display cleared, and the document itself displayed. Finally, the initial position report of the window is sent, and the DocTracker's event handler is added to the document's list of handlers.

5.2. Dynamically Displaying Results

The Python usage statistics display program is used as an embedded applet, drawing a bar chart of current usage statistics into the area of the document in which it is placed. As new usage information becomes available, the displayed bar chart is updated dynamically. Two classes make up the applet, StatsCollector and ihEmbed, as shown in Figure 5.3. The former is a small class that processes the data sent from the usage server, transforming it into an array of bar data. The latter is the actual embedded object, containing all the code for initializing the applet, creating the StatsCollector, and actually drawing the bar chart.

As with the usage retrieval program, the display program first initializes information about the default host and port of the usage server, and retrieves the modules it will use. In addition, it defines two values representing the state of the StatsCollector parser.

Again like the usage retrieval program, the display program's StatsCollector class is a subclass of the iHTML LineURL class. Upon initialization, it sets up the variables it will use to store the current bar values and initializes its superclass to start the URL transaction. After that has been accomplished, it sends a "`* watch`" command to the usage server, telling the server that it will be watching the server's usage information.

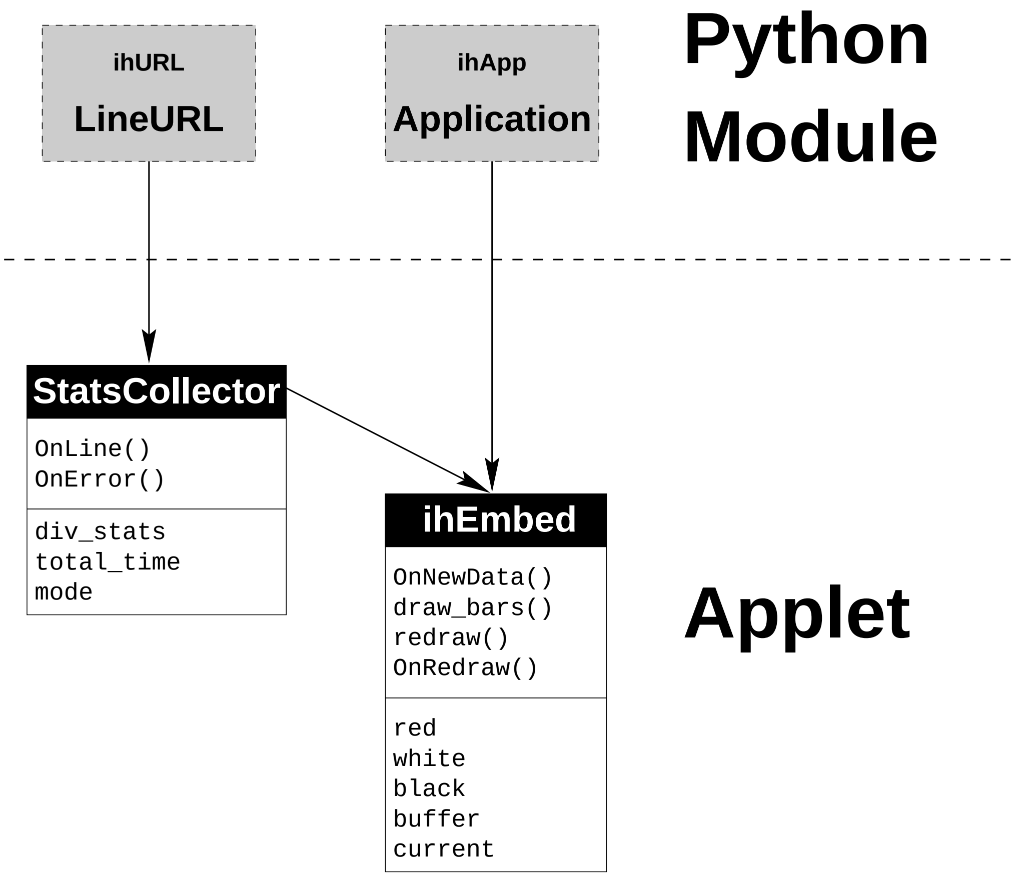


Figure 5.3 Architecture of the usage display program.

The StatsCollector class's `OnLine()` method is responsible for parsing the usage information it receives from the server. When the class first is instantiated, it starts out in `MODE_READY` state. Upon receiving a "> `begin view`" command, it switches to `MODE_VIEW` state and begins processing lines of view information. Any "> `total`" and "> `div`" lines it receives in view mode are parsed into its new bar information. Finally, when it receives a "> `end view`" command, it switches back to its `MODE_READY` state and calls `OnNewData()` to display the new bar chart. The `OnNewData()` method must be defined by a subclass.

The ihEmbed class inherits from both the iHTML Application and the StatsCollector classes. Its initialization method sets up its display area, then creates the colors and off-screen buffer it will use for rendering. The method then parses its arguments for the URL of the server, just as in the usage retrieval program. It then computes various dimension information from the font it is using, which will later be used in rendering the bars. Finally, it calls the initialization method of its StatsCollector parent class to start a transaction with the usage server.

The `OnNewData()` method, called by the StatsCollector when new usage data is available, is responsible for redrawing and displaying the bar chart. Using multiple inheritance with the ihEmbed class, this method automatically defines the StatsCollector's `OnNewData()` method, while still having direct access to the Application class's widget information needed to do the actual rendering. It essentially massages the raw bar data into the pixel dimension and position information used in rendering the bars, then calls a method that does the actual drawing.

The `draw_bars()` method uses the information computed in the previous method to draw the corresponding bar chart into an off-screen buffer. To display the actual buffer, the `redraw()` method copies it into the widget's display area and then forces the system to finish all drawing operations. A final method, `OnRedraw()`, handles widget redraw events, which simply result in the off-screen buffer being copied back into the widget.

6. LANGUAGE DEVELOPER INTERFACE

An iHTML language module implements the code needed to handle a particular file type—whether it be a traditional media type (such as a GIF image or MPEG movie) or an interpreter for a scripting language. Each language module exists as an external piece of machine code (on most platforms, a shared library) that is compiled from C source code. The iHTML system loads and executes a language module when it encounters a data type that the module can handle.

This chapter describes the low-level iHTML architecture, from the perspective of a developer implementing a new language module. Section 6.1 provides an introduction to some general design points of the system. Since the public interfaces to the browser and library components of the system are used extensively in the development of a language module, a brief overview them is presented in Section 6.2. A final section goes into detail on the interface that the language module itself must implement.

6.1. Introduction

The iHTML architecture is defined in terms of a set of C language types, structures, and functions. There are three main classes of symbols that iHTML defines within the global C language name space.

The interface to the **Browser Component** is defined through a set of functions and types that are prefixed with "`BR_`" and "`IH`", respectively. Examples are "`BR_Reformat()`" and "`IHWidgetRep`". (Note, however, that there are a few exceptions in the type names.)

The interface to the **iHTML Library Component** is defined through a set of functions and types prefixed with "`IH_`" and "`IH`", respectively. In addition, actual structure names use the prefix "`ih_`". Examples are "`IH_AllocBuffer()`", "`IHModuleInfo`", and "`ih_module_info_rec`".

The interface to the **Language Component** consists of type names prefixed with "`IH`". The functional interface to language modules is through a structure, which has its own name space; these function names do not use any prefix.

6.1.1. Language Modules and MIME Content Types

The iHTML system must have some way to map between the MIME content type of a file it receives and the language module(s) that can understand that data. There are two steps in this mapping: converting a content type into the name of the language that can handle it, and determining where that language's module can be found in the computer's file system.

IHTML does not define directly where language modules are located. Instead, it specifies that each module must be mapped to a unique name, leaving it up to the browser component to map the abstract names to the underlying file system. The mapping is accomplished through a set of browser component routines (`BR_GetBasePath()` and `BR_ParsePath()`) that are invoked with the name of a language and return a set of directories that make up that language's search path. When constructing the path, the browser can take into account other information, such as the machine architecture it is running under (to support multiple architectures in the same file space). The system then looks through these directories for a shared library constituting the language module it needs.

The iHTML library can also request that the search path be for the "NULL" language. This signifies the top-level path, in which all languages are defined. When the system is first initializing itself at program startup time, the iHTML library requests the "NULL" path and searches it for one or more files that contain mappings between language names and content types. There is usually one such file for each language in the system. Some examples of mapping files are listed below.

- A language module that displays MPEG movies would have a content type mapping file similar to the following:

  ```
  Name: mpeg
  Content-type: video/x-mpeg
  Content-type: video/mpeg
  ```

  This tells the browser to look in the path of a language named "mpeg" for a language-module that can display files encountered with either of the given content types.

- Similarly, a language module that displays textual data can be defined as:

  ```
  Name: text
  Content-type: text/*
  ```

  Here, the system interprets the "*" as matching anything (the same syntax used by metamail).

- A language module for a full scripting language might be defined as:

  ```
  Name: python
  Content-type: application/x-ihtml; language=python
  ```

  The browser will map all files with both a content type of "`application/x-ihtml`" and a parameter named "`language`" whose value is "`python`", to the python module.

As a more complete example, the file system organization of iHTML language modules under Unix is usually something like the following:

```
-rw-------   1 hackbod  grads         64 Sep 30 23:34 python.lang
drwx------   4 hackbod  grads       1024 Sep 30 23:34 python/
-rw-------   1 hackbod  grads       1094 Sep 30 23:34 python/COPYRIGHT.python
drwx------   2 hackbod  grads       1024 Sep 30 23:34 python/hp-ux
-rwx------   1 hackbod  grads    1207916 Sep 30 23:34 python/hp-ux/impl.sl*
drwx------   2 hackbod  grads       1024 Sep 30 23:34 python/aix
-rwx------   1 hackbod  grads    1485342 Sep 30 23:34 python/aix/impl.so*
-rw-------   1 hackbod  grads         32 Sep 30 23:34 text.lang
drwx------   4 hackbod  grads       1024 Sep 30 23:34 text/
-rw-------   1 hackbod  grads       1094 Sep 30 23:34 text/COPYRIGHT.text
drwx------   2 hackbod  grads       1024 Sep 30 23:34 text/hp-ux
-rwx------   1 hackbod  grads     102680 Sep 30 23:34 text/hp-ux/impl.sl*
drwx------   2 hackbod  grads       1024 Sep 30 23:34 text/aix
-rwx------   1 hackbod  grads     134320 Sep 30 23:34 text/aix/impl.so*
```

The directory tree has two language modules, `python` and `text`, both with implementations available for HP/UX and AIX machines (`impl.sl` and `impl.so`). The files `python.lang` and `text.lang` describe the content types each language can handle.

6.2. iHTML Programming Interface

In the course of implementing a language module, the language developer makes use of many of the types and functions defined by the browser and iHTML library components. These two components essentially define the programming interface for the language module, and thus are discussed here prior to the language module itself. In addition, the iHTML library component defines a number of types that either must be completely implemented, or extended as part of a language module implementation. We first discuss the interface to the browser component and then cover the iHTML library.

6.2.1. Browser Services

As discussed in Section 3.2.1, the browser component's services are divided into seven broad categories. **HTML Parse Tree** services include functions that provide access to, and manipulation of, the browser's internal HTML parse tree representation. Its **Document** services provide global control of the current HTML document being displayed and the browser itself. For parsing and retrieving URLs, the browser includes a set of **URL** services. User interaction is divided into two further categories: **Event** services define user interface and system events that inform the rest of the system of user activity, while **User Interface** services are used for creating and interacting with a graphical user interface. Operations for loading, examining, and unloading OS-level shared libraries (i.e., language modules) are supplied through the **Shared Library** services. Finally, there are various **Miscellaneous** services for other routines operations.

HTML Parse Trees

A browser's **HTML Parse Tree** services include one type and a set of functions for accessing a browser's internal parse tree representation. They allow language modules not only to construct HTML documents dynamically, but also to examine and modify existing documents. Table 6.1 shows the browser functions that are a part of the parse tree services. The HTMLNode type is the object through which other components in the iHTML system interact with a browser's internal parse tree. It represents a single node in that tree (e.g., a tag, block of text, comment, etc.). All interaction with the parse tree is through the listed functions, which are given opaque pointers to HTMLNode objects.

Function	Description
BR_AllocMarkup()	Allocates a new HTMLNode with selected type and attributes
BR_ChildMarkup()	Returns the HTMLNode that is the child to the given HTMLNode, essentially traversing deeper into a parse tree
BR_CutMarkup()	Deletes a range of HTMLNodes in a tree, returning the first HTMLNode that has replaced them
BR_FirstMarkup()	Returns the absolute first HTMLNode in the given tree, i.e. the root of the tree, which should always be `<HTML>`
BR_FreeMarkup()	Deallocates a tree of HTMLNode objects
BR_FreeMarkupAttr()	Deallocates the memory associated with an attribute value that was previously returned by `BR_MarkupAttr()`
BR_GetTagID()	Returns the ID number the browser uses for a particular tag string (e.g., "H1")
BR_InitMarkup()	Initializes a raw HTMLNode tree returned by the browser's parser
BR_InsertMarkup()	Inserts a single HTMLNode into an existing tree
BR_InsHeadMarkup()	Places a tree of HTMLNodes as the first child of an existing HTMLNode
BR_InsTailMarkup()	Places a tree of HTMLNodes as the last child of an existing HTMLNode
BR_LastMarkup()	Returns the absolute last HTMLNode in the given tree, i.e. the very last leaf node of the tree
BR_MarkupAttr()	Returns a string containing the value of a selected attribute in an HTMLNode
BR_MarkupLanguage()	Returns the interface to the language module that owns a particular HTMLNode
BR_MarkupOwner()	Returns the language-side object that owns a particular HTMLNode
BR_MarkupText()	Returns a string of ASCII text associated with an HTMLNode
BR_MarkupType()	Returns a browser ID indicating the type of an HTMLNode
BR_NextMarkup()	Returns the HTMLNode that occurs logically after a given HTMLNode
BR_NextMarkupDepth()	Returns the HTMLNode that would occur next in a tree, if a depth-first traversal were being performed
BR_NextMarkupType()	Performs a depth-first search starting at a given place in an HTML tree, looking for the next node with the selected type
BR_ParentMarkup()	Returns the HTMLNode that is the parent of a given node, essentially traversing up out of a parse tree
BR_ParseHTML()	Convert a string of HTML markup into its HTMLNode tree representation
BR_PasteMarkup()	Places a tree of HTMLNodes into another tree
BR_PrevMarkup()	Returns the HTMLNode that occurs logically before a given HTMLNode
BR_PrevMarkupDepth()	Returns the HTMLNode that would occur previously in a tree, if a depth-first traversal were being performed
BR_RemoveMarkup()	Deletes a selected HTMLNode

Table 6.1 Functions in a browser's HTML parse tree services

Function	Description
`BR_AllocResources()`	Called by the system when starting a new document interface, this function is used to allocate any resources it needs to associate with that interface
`BR_FreeResources()`	Deallocates and cleans up any browser-side resources that were allocated by `BR_AllocResources()`
`BR_GetDocHTMLTree()`	Returns the root `HTMLNode` of the parse tree currently being displayed by an IHDocument
`BR_Reformat()`	Examines and re-formats a document's current HTML parse tree, displaying it to the user
`BR_ReportError()`	Reports an error condition to the browser user
`BR_SetDocHTMLTree()`	Sets an HTML parse tree as the text currently being displayed by a IHDocument
`BR_ShowLink()`	Displays to the browser user information about a hypertext link within the browser
`BR_ShowProgress()`	Displays to the browser user a progress report for a running operation

Table 6.2 Functions in a browser's document services

Documents

A browser's document services include functions for examining and modifying the global state of a document through its HTML parse tree, reporting information about the document, and managing the browser's interaction with the rest of the iHTML system. These services make use of two data types. The IHBrowser is a handle on the display context of a document, typically a user interface window the browser is managing. The document itself is represented by an IHDocument type. It is through the IHDocument that the system gains access to a document's HTML parse tree for examination and manipulation, and performs other document-level operations. The functions defined as part of the document services are shown in Table 6.2.

Function	Description
`BR_DerefRetrieveRequest()`	Removes a reference from an IHRetrieveRequestP, deallocating it if that was the last reference on it
`BR_GetRequestBuffer()`	Gets the current data associated with an IHRetrieveRequestP
`BR_GetRequestHeader()`	Returns a string representing the value of a header associated with an IHRetrieveRequestP (e.g., the MIME "`Content-type`" header)
`BR_GetRequestStatus()`	Returns the current IHReqStatus code associated with an IHRetrieveRequestP
`BR_GetRequestURL()`	Get a string representing the fully expanded URL that an IHRetrieveRequestP is retrieving
`BR_RetrieveURL()`	Create a new IHRetrieveRequestP
`BR_SetRequestBuffer()`	Sets the data associated with an IHRetrieveRequestP
`BR_WriteRequestBuffer()`	Send data to the remote side of an IHRetrieveRequestP

Table 6.3 Functions in a browser's URL services

URLs

A browser's URL services include the types and functions needed to access the browser's internal networking and URL handling mechanisms. The primary data type used in these services is the IHRetrieveRequestP, which is an abstract handle on an active URL transaction. As data becomes available from the URL, a function of type IHRetrieveCallback is called to inform the program that initiated the URL request. Finally, an enumeration of status codes is defined by the IHReqStatus type. Table 6.3 lists the functions that are included in these services.

Events

A browser sends information about user actions and activity to other parts of the iHTML system through its event services. A number of functions are part

Function	Description
`BR_AllocEvent()`	Creates a new IHEvent object
`BR_DupEvent()`	Creates a new IHEvent object, based on an existing one
`BR_FreeEvent()`	Deallocates a previously created IHEvent object
`BR_FreeEventAttr()`	Deallocates an attribute that was previously retrieved from an IHEvent
`BR_GetEventAttr()`	Retrieves the value of an attribute associated with an IHEvent
`BR_SetEventAttr()`	Sets the value of an attribute associated with an IHEvent

Table 6.4 Functions in a browser's event services

of these services, as shown in Table 6.4. In addition, the IHEvent type defines the public data contained in an event, including the scripting interface, language environment, and document with which it is associated. Each event also includes two integer identifiers, representing its **class** and **code**. The event class is the major type of the event. It may be one of two values, with **document events** being those that are associated with a browser document, and **widget events** being those produced by a user interface widget inside a document. An event's code is the actual action that occurred. For document events, the code may indicate that the view on the document changed, the user moved over a hypertext link, etc. Widget event codes include mouse movement, keyboard events, etc. Events also have **attributes** associated with them. An attribute provides more detailed information about the action that occurred, such as the mouse position, key pressed, or URL of an anchor.

User Interface

There are a large number of browser services dedicated to the creation and manipulation of graphical user interfaces. Table 6.5 summarizes the functions that are a part of these user interface services. The services center around the IHWidgetRep type, which represents a browser-side user interface object. The object encapsulates a graphical area in a document, with its own display context for drawing graphics and receiving widget class IHEvent reports. A browser's widgets are organized in an **instance hierarchy**, which defines the graphical relationship between them. The children in a widget use a coordinate system relative to their parent (e.g., the location (0,0) would position a child at the top-left corner of its parent widget).

An actual display context for drawing operations is defined by the IHPixmap type, which can either be the drawing space of a IHWidgetRep or a separate off-screen buffer. Similar to the IHEvent type's attributes, widgets and pixmaps have a number of associate **member** values, used to get and set the state information associated with a particular widget or pixmap. Members defined by the system include such things as a pixmap or widget's dimensions, current drawing pen, and the document with which it is associated.

All drawing operations go through an IHPixmap object and make use of two additional types. The IHColor represents a pen color for rendering, composed of a triple representing that pen's red, green, and blue components. An IHFont is a handle on a text font, which determines the shape and style of the font used in text drawing operations. Finally, an IHFutureCall represents the context for a function call that is to occur at some later time, and is often used for such things as animation.

Shared Libraries

Access to the underlying operating system's operations for loading and using shared libraries is provided through a browser's shared library services. The functions included in these services are summarized in Table 6.6. They make use of the

Function	Description
BR_AddFutureCall()	Constructs the context on a function call that is to be performed at some future time, and returns a new IHFutureCall object for it
BR_AllocIHColor()	Allocates a new IHColor object that can be used to set the pen color for future drawing operations
BR_AllocIHPixmap()	Creates a new IHPixmap for an off-screen rendering area
BR_AllocWidget()	Creates a new IHWidgetRep object, making it the child of an existing IHWidgetRep
BR_FlushDisplay()	Finishes all drawing operations that are currently in progress
BR_FreeIHColor()	Deallocates an IHColor that was previously allocated with BR_AllocIHColor()
BR_FreeIHPixmap()	Deallocates an IHPixmap that was previously allocated with BR_AllocIHPixmap()
BR_FreeWidget()	Deallocates an IHWidgetRep that was previously allocated with BR_AllocWidget()
BR_GetPixmapMembers()	Retrieves one or more member values associated with an IHPixmap
BR_GetWidgetMembers()	Retrieves one or more member values associated with an IHWidgetRep
BR_PixBltPixmap()	Copies the raster data from one IHPixmap into another
BR_PixClearRectangle()	Erases a rectangular region in an IHPixmap
BR_PixDrawArc()	Draws an unfilled arc in an IHPixmap
BR_PixDrawLine()	Draws a line in an IHPixmap
BR_PixDrawRectangle()	Draws an unfilled rectangle in an IHPixmap
BR_PixDrawText()	Draws a string of text in an IHPixmap
BR_PixFillArc()	Draws a filled arc in an IHPixmap
BR_PixFillRectangle()	Draws a filled rectangle in an IHPixmap
BR_PixSetBackPen()	Sets the background pen used in future drawing operations in an IHPixmap
BR_PixSetFont()	Sets the text font used in future drawing operations in an IHPixmap
BR_PixSetForePen()	Sets the foreground pen used in future drawing operations in an IHPixmap
BR_PixTextExtent()	Measures the dimensions that a string of text would cover, if it was drawn into an IHPixmap
BR_RemFutureCall()	Aborts an existing IHFutureCall that hasn't yet happened
BR_SetPixmapMembers()	Sets one or more member values associated with an IHPixmap
BR_SetWidgetMembers()	Sets one or more member values associated with an IHWidgetRep

Table 6.5 Functions in a browser's user interface services

Function	Description
`BR_GetSharedSymbol()`	Retrieves the symbol with a given name from an IHSharedModule
`BR_LoadSharedModule()`	Loads a shared library into the browser's address space
`BR_UnloadSharedModule()`	Removes a shared library that was previously loaded with `BR_LoadSharedModule()`

Table 6.6 Functions in a browser's shared library services

IHSharedModule type, which represents a handle on a shared library that has been loaded into the system.

Miscellaneous

There are a number of miscellaneous functions and types that don't fit into any of the previously described categories. Included in this miscellaneous category are functions for handling search paths, accessing files, performing string comparisons, and retrieving version information. The functions included in these services are summarized in Table 6.7. In addition, the type IHDirScan is used in a number of functions for examining the contents of a file system directory.

6.2.2. iHTML Library Services

As discussed in Section 3.2.2, the iHTML library component's services are divided into five broad categories. **Document Interface** services manage the interaction between browser documents and the scripts running on them. Each script's actual back-end language module is handled through the **Language Module** services. The library component includes a set of **Buffer** and **Doubly-linked List** services, for manipulating blocks of byte data and linked list structures. Finally,

Function	Description
BR_BeginDirScan()	Starts a session retrieving the contents of a file system directory
BR_EndDirScan()	Finishes a directory scanning session
BR_FreeParsePath()	Deallocates the memory returned by BR_ParsePath()
BR_GetBasePath()	Returns an abstract string defining the base search path for the browser
BR_GetCapability()	Returns information about the browser's capabilities, such as whether it supports tables or graphics
BR_GetVersion()	Retrieves version information about the browser
BR_NextDirScan()	Moves to the next entry in a directory scanning session
BR_OpenDirFile()	Opens a file in a particular directory, and returns a C file handle for it
BR_OpenTempFile()	Creates a temporary file and returns a C file handle for it
BR_ParsePath()	Transforms a base search path into a list of directories for a particular language
BR_StrCmp()	Performs a case-insensitive string comparison

Table 6.7 Functions in a browser's miscellaneous services

there are **Miscellaneous** services for various operations that do not fall into one of the previous groups.

Document Interface

The document interface services include a number of data types used to store high-level scripting information related to a document (e.g., the `<OBJECT>` tags embedded in a document), and functions for managing the interaction between these data structures and the browser itself. Table 6.8 summarizes the functions that are a part of the document services. At the highest level is the IHGlobalInfo structure, which stores the context associated with one browser document's entire scripting session. It includes such information as any document script, embedded applets, and language modules that are active in the document.

Function	Description
`IH_FreeInterface()`	Deallocates an IHGlobalInfo structure and all resources attached to it
`IH_FreeParam()`	Deallocates the information for a `<PARAM>` tag
`IH_ParseEmbedded()`	Parses an HTML `<OBJECT>` definition, creating a new IHEmbeddedInfo object for it
`IH_ParseInterface()`	Parses an HTML parse tree, allocating and initializing a new IHGlobalInfo for it
`IH_StartInterface()`	Re-starts the execution of all scripts in a IHGlobalInfo context
`IH_StopInterface()`	Stops the execution of all scripts in a IHGlobalInfo context

Table 6.8 Functions in the iHTML library's document services

There are three additional data types associated with an IHGlobalInfo object. Every language module that has one or more scripts executing in the document is allocated an IHLangEnv structure, which contains script context information for the entire document. A document script is represented by an IHModuleInfo structure, and each applet is represented by an IHEmbeddedInfo structure. These two types are based on a common structure, IHCodeInfo, which stores general scripting information such as the language module executing the script, the content type and actual data for a script that was retrieved over the network, and information about any `<PARAM>` tags that were included with its `<OBJECT>` tag.

Language Modules

The iHTML library component is responsible for the management of all back-end language modules, as well as providing a general interface to them for the rest of the system. These services are provided through the functions summarized in Table 6.9. In addition, the library component defines a standard interface that a language module must provide to the system. The language module interface

Function	Description
`IH_AllocWidget()`	Creates a new language-side widget object in a particular language environment
`IH_CleanHTMLNode()`	Cleans up any language-side information that has been associated with a particular HTMLNode, before deallocating it
`IH_DerefMarkup()`	Decrements the reference count on a language-side HTML parse tree
`IH_FreeWidget()`	Deallocates a new language-side widget object in a particular language environment
`IH_GetWidgetMembers()`	Retrieves the member values of a language-side widget object
`IH_HandleEvent()`	Sends an event off to the language module that can handle it
`IH_HeadMarkupNode()`	Returns the first HTMLNode that is being managed by a language-side HTML parse tree
`IH_ImportLanguage()`	Returns the IHLanguage handle for a language module that can handle a particular content type, loading and initialize the language module if needed
`IH_RefMarkup()`	Increments the reference count on a language-side HTML parse tree
`IH_ReleaseLanguage()`	Releases an IHLanguage handle that was previously returned by `IH_ImportLanguage()`
`IH_SetMarkupHead()`	Informs a language-side HTML parse tree that it has a new head HTMLNode
`IH_SetWidgetMembers()`	Sets the member values of a language-side widget object
`IH_TailMarkupNode()`	Returns the last HTMLNode that is being managed by a language-side HTML parse tree

Table 6.9 Functions in the iHTML library's language services

revolves around the IHLanguage type, which is used as a handle on a language module. The complete language module interface is discussed in more detail in Section 6.3.

Function	Description
`IH_AllocBuffer()`	Creates a new IHBufferP, optionality providing it with an initialize size and/or data
`IH_AppendBuffer()`	Adds new data to an existing IHBufferP
`IH_FreeBuffer()`	Deallocates an existing IHBufferP

Table 6.10 Functions in the iHTML library's buffer services

Buffers

The iHTML library component includes support services for creating and managing arbitrary blocks of byte data. They are centered around the IHBufferP type, which stores a block of byte data of arbitrary length. Manipulation of the buffer is provided through the functions summarized in Table 6.10. The data in a buffer is accessed by examining the type's `length` and `buffer` fields.

Doubly-linked Lists

The iHTML library component's doubly-linked list services are a pair of standard types and several functions for managing arbitrary doubly-linked list structures. The two types are IHList and IHNode. The former is a handle on a complete list, holding information about the first and last nodes in the list. To make use of the list services, an IHNode structure is overlaid on a structure that will be placed in the list; that structure can then be manipulated as a node, using the functions summarized in Table 6.11.

Miscellaneous

There are a number of miscellaneous functions and types that don't fit into any of the categories described previously. The most important of these are the

Function	Description
`IH_AddHead()`	Inserts an IHNode into the front of a list
`IH_AddTail()`	Appends an IHNode to the end of a list
`IH_AllocNode()`	Allocates and initializes a new IHNode structure
`IH_AllocList()`	Allocated and initializes a new IHList structure
`IH_FreeList()`	Deallocates an IHList structure, along with any nodes attached to it
`IH_GetHead()`	Retrieves the first node in a list
`IH_GetTail()`	Retrieves the last node in a list
`IH_InitList()`	Initializes an existing IHList structure
`IH_InsNode()`	Inserts an IHNode into a particular location in a list
`IH_NextNode()`	Retrieves the IHNode that occurs immediately after a given node
`IH_PrevNode()`	Retrieves the IHNode that occurs immediately before a given node
`IH_RemHead()`	Removes the head node from a list
`IH_RemNode()`	Removes an arbitrary IHNode from a list
`IH_RemTail()`	Removes the tail node from a list

Table 6.11 Functions in the iHTML library's doubly-linked list services

content type services, which are used to parse and compare MIME "`Content-type`" headers. A parsed content type is expressed by the IHContentType type, which directly represents the type, subtype, and arguments. In addition, there are services for transforming strings into the ID codes used internally by the system, retrieving version information, and initializing or terminating the system. Table 6.12 summarizes the functions included in miscellaneous services.

6.3. Language Module Interface

A language module exists as an external piece of code (usually a shared library) that the iHTML system dynamically loads to handle a particular type of data it has encountered. In its implementation a language module may call any functions discussed in Section 6.2, in order to make use of the services provided

Function	Description
`IH_FindNameEntry()`	Looks up the system's ID code for a string
`IH_FreeContentType()`	Deallocate the resources associated with an IHContentType
`IH_GetVersion()`	Retrieves version information about the iHTML library
`IH_Initialize()`	Initializes the iHTML library and scans the system for language modules
`IH_MatchContentType()`	Compares two IHContentType objects
`IH_ParseContentType()`	Converts a MIME "`Content-type`" string into its equivalent IHContentType object
`IH_Terminate()`	Deallocates all global resources used by the iHTML library

Table 6.12 Functions in the iHTML library's miscellaneous services

by the rest of the system. In addition, it must implement its own public interface, which the iHTML system uses to access the language module's own services.

The primary interface between a language module and the rest of the iHTML system is defined by the IHLanguage type. An object of this type not only provides access to its language module services, but also serves as a handle on that language module. In particular, the iHTML library's `IH_ImportLanguage()` function returns an IHLanguage object as a representation of the language module it imported, a handle that is used later to tell the system when the language module no longer is needed. Table 6.13 shows the fields of the IHLanguage type. They include such information as the version of the module, build date, and name. The `Fn` field is a pointer to another structure, IHLanguageFunc, which is a series of pointers to functions defining the module's functional interface (the functions are described in later subsections).

The iHTML system must have some way to retrieve a language module's IHLanguage object. Thus the public symbol table of a language module consists

Field	Description
`Fn`	Functional interface to the language module
`Version`	Integer version number of the language module
`Revision`	Integer revision number of the language module
`Submake`	Integer minor make build number of the language module
`BuildDate`	String containing the date and time the language module was created
`Name`	String containing the name of the language or data type implemented by the module (e.g., "python")
`LangEnvSize`	Size in bytes of the IHLangEnv type used by the language module

Table 6.13 Fields defined by the IHLanguage structure

of one function, which is called by the iHTML system to initialize a module after loading it and returns the module's IHLanguage interface. iHTML looks for the function under two names. First, it tries "`IHTML_`*language*`()`", where *language* is the (case-sensitive) name of the module, such as "`python`". If unable to find a function of that name, the system then tries the generic name "`IHTMLinterface()`". In either case, the function is defined as type LanguageInit.

The rest of this section discusses the actual functions in a language module's IHLanguageFunc structure. The functions are divided into four general categories. First are the **HTML Parse Tree** functions, which create and manage the language-side representation of the HTML parse tree. A language module has context information, associated with each document in which it has scripts executing, which is managed by its **Environment** functions. The management and execution of scripts themselves is the responsibility of **Script** functions. Finally, **User Interface** functions are responsible for creating and managing the language-side representation of user interface objects.

Function	Description
`IHMarkup_Deref()`	Removes a reference from a given language-side IHMarkup object
`IHMarkup_HeadNode()`	Retrieves the first HTMLNode that is owned by an IH-Markup object
`IHMarkup_Ref()`	Adds a reference from a given language-side IHMarkup object
`IHMarkup_SetHead()`	Sets the first HTMLNode that is owned by an IHMarkup object
`IHMarkup_TailNode()`	Retrieves the last HTMLNode that is owned by an IH-Markup object

Table 6.14 Functions in a language module's HTML parse tree services

6.3.1. HTML Parse Trees

A language module's HTML parse tree services allow it to keep track of browser-side HTMLNode objects it has allocated. The services revolve around one type, IHMarkup, which is the language-side handle on a tree of HTMLNode objects. One IHMarkup object may be associated with a single HTMLNode or an entire tree. As trees owned by different IHMarkup objects are pasted together and taken apart, the iHTML system helps each language module keep track of a *reference count* on its IHMarkup objects, so that they may be deallocated when no longer needed. The actual implementation of an IHMarkup type is language-dependent—the rest of the system simply treats it as an opaque type, manipulated through the language module functions listed in Table 6.14.

6.3.2. Environment

A language module must keep track of context information associated with each document in which it is being used. The IHLangEnv type is a handle on a

Function	Description
`AllocLangEnv()`	Allocates the language module resources associated with an IHLangEnv
`ExecLangEnv()`	Starts global script execution in a language environment
`FreeLangEnv()`	Deallocates the language module resources associate with an IHLangEnv
`StopLangEnv()`	Stops global script execution in a language environment
`Term()`	Deallocates all global resources used by a language module, prior to unloading it from the system

Table 6.15 Functions in a language module's environment services

single execution environment in a language module. There is one IHLangEnv object allocated for each document in which the language module is active. It serves as an execution context for the high-level document script and for interaction between embedded scripts. While the iHTML system defines a number of fields that must appear at the beginning of a language module's IHLangEnv type, the module is free to extend the type to include its own private data. Since the language module does not itself allocate the IHLangEnv type, it places the size of the type it uses in the `LangEnvSize` field of its IHLanguage object, telling the system how many bytes to allocate. Table 6.15 lists the language module functions that the rest of the system then uses to manipulate the language's IHLangEnv objects.

6.3.3. Scripts

Within a document's language environment, the language module must provide services for manipulating the actual scripts it contains. The functions included in the script services are listed in Table 6.16. Because the IHEmbeddedInfo and IHModuleInfo types are not defined by the language module itself, it needs some way to attach its own context information to objects of those types. Thus a lan-

Function	Description
`AllocEmbedded()`	Allocates the language-side resources associated with an embedded applet
`AllocModule()`	Allocates the language-side resources associated with a document script
`FreeEmbedded()`	Deallocates the language-side resources associated with an embedded applet
`FreeModule()`	Deallocates the language-side resources associated with a document script
`StartEmbedded()`	Starts the execution of an embedded applet
`StopEmbedded()`	Stops the execution of an embedded applet

Table 6.16 Functions in a language module's script services

guage module's script context is defined through the IHLangCode type, an opaque type containing information about a single piece of executing code. The language module can define the IHLangCode type in any convenient way, and usually uses it to store information such as the compiled form of a script, a symbol table for the script, etc. The resulting object then is attached to the script's IHEmbeddedInfo or IHModuleInfo when `AllocEmbeded()` or `AllocModule()` is called, respectively.

6.3.4. User Interface

The final services implemented by a language module are those for managing its user interface. While the browser component defines the IHWidgetRep object for drawing graphics and retrieving user events, the object does not include any intelligence for determining what to draw or how to respond to the user. Instead, the language module is responsible for implementing an IHWidgetObj type, which it attaches to a IHWidgetRep to tell it what to do. In addition, the user interface

Function	Description
`HandleEvent()`	Dispatches an event to the appropriate IHWidgetObj or document script in a language environment

Table 6.17 Functions that are part of a language module's user interface services

services implement the single function shown in Table 6.17. The other components of the system use the function to send events to the language module.

7. IMPLEMENTATION DETAILS

The iHTML system has been incorporated into NCSA's X Mosaic browser. There are two language modules written for it, one displaying plain ASCII text data and another encapsulating an interpreter for the Python language. This chapter examines some of the issues in the C language implementation of these components. It first looks at the iHTML implementation itself, then the interface between the library and the X Mosaic browser, and finally the two language modules.

7.1. iHTML Library Implementation

The public interface to all the iHTML components is defined by the iHTML library. This is done in the header file `IHScript.h`, which the code for all components must include in order to access the system. The file is composed primarily of function and type definitions. Types that are to be privately implemented by other components (such as the browser's IHWidgetRep) are defined as stubs (i.e., void pointers). Components that use this header file may selectively override the type definitions, in order to access the private data.

The implementation of this component is divided into nine functional units. The majority of them are defined in the files `IHLangs.c` and `IHScript.c` (covered in sections 7.1.1 and 7.1.2, respectively). The next largest is `IHdlist.c`, which implements the general-purpose doubly-linked list type that is used throughout the rest of the system. `IHBuffer.c` implements the routines for creating and managing the IHBufferP data type. `IHEntry.c` defines mappings from various ID numbers used in the iHTML system (e.g., event class and code IDs) and their equivalent textual names. `IHMarkup.c` and `IHWidget.c` implement the iHTML library's markup and widget routines, respectively. Finally, `IHMisc.c` defines various other routines, such as library initialization and termination, retrieval of version information, etc.

7.1.1. Managing Language Modules

The `IHLangs.c` file implements the functionality for managing language modules in the system. Part of its functionality involves the processing of MIME "Content-type" headers, and performing comparisons between them. The textual content type is parsed into an IHContentType structure, which explicitly represents the type, subtype, and arguments that the textual representation encodes. Two IHContentType structures can then be matched against each other to determine whether one is equivalent to another.

When the iHTML system is first initialized, it scans through its root search path for language modules in the system. It determines the available language modules by looking for language definition files, which end with the prefix "`.lang`" and define mappings from MIME "Content-type" headers to the modules that handle them. In the process finding and parsing these files, a list is created of all the languages found; each node in the list includes information about the name of the language, where it was found, and a list of all the content types it implements.

As the system is running, it maintains another list of all the language modules that are currently loaded. When a language to handle a particular content type is sought, this list is examined first in order to make use of any language that is already loaded. If one is found, it is used directly. Otherwise, the system searches through the full list of available modules, looking for one that has a match for the desired type. If found, the browser component's shared module routines are used to load the module, it is initialized, and a new node for it is placed on the loaded list.

This top level functionality is made publicly available through the `IH_ImportLanguage()`, `IH_LockLanguage()`, and `IH_ReleaseLanguage()`. Each loaded module has a *reference count* associated with it. Importing or locking that language increases the count, while releasing it decreases the count. When the count goes to zero, the language module is automatically terminated and removed from memory.

7.1.2. Document Scripts and Applets

The `IHScript.c` file implements the main functionality of the iHTML library. This involves management of the IHGlobalInfo structure, which is the iHTML-side document context that keeps track of the language environments, document scripts, and applets that are associated with a single HTML document. The primary interface to this file is through the functions `IH_ParseInterface()`, `IH_StartInterface()`, and `IH_ParseEmbedded()`. The first two are used to manage the top-level document context, while the third manages the context of individual applets appearing in the document.

Figure 7.1 shows an overview of how the IHGlobalInfo structure is organized. It includes three lists, containing the language environments, document scripts, and applets associated with the document. There is one language environment for every language executing in the document. When a new document script or applet is added to the structure, the system checks to see if a language environment already exists for the code; if not, a new one is created and linked in to the structure. In addition, the IHModuleInfo and IHEmbeddedInfo structures share a common element, the IHCodeInfo structure. By using the same structure, the code for managing and associating the scripts can be shared between them.

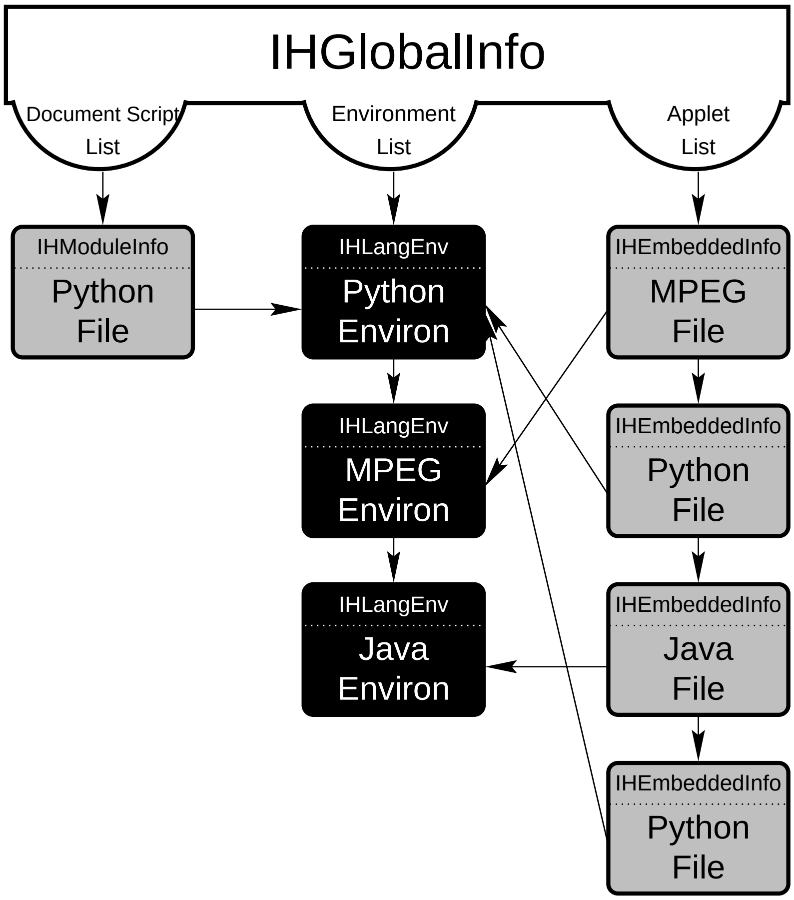


Figure 7.1 Organization of the iHTML library's IHGlobalInfo structure

When the browser first retrieves a new HTML document, it sends the document to `IH_ParseInterface()`. This function examines the document, determining if there are any applets or a document script associated with it. If so, it creates a new IHGlobalInfo environment for the document. In addition, if it finds a document script, it requests that the browser begin retrieving the script from the network and adds a IHModuleInfo node to the document context. The function finally returns the created IHGlobalInfo.

As the browser parses and renders the document, it calls this file's `IH_ParseEmbedded()` function for each `<OBJECT>` tag it encounters. The function parses the tag, begins a network request through the browser for the object, and adds a IHEmbeddedInfo node to the document context. It also associates the new script with a browser-side IHWidgetRep in which it will appear.

The browser may call the `IH_StartInterface()` function at any time after it has first created the document context. The function begins execution of all scripts associated with the document. It first runs through the list of language environments and initializes their execution state, which has the effect of starting any document scripts in them. Next, it runs through each applet in the document and starts their execution. The procedure is more complicated if a document script has not yet been retrieved over the network, since a language environment cannot start executing until all its document scripts are available. To take this situation into account, the system creates an IHDelayedStartup structure for every applet whose language environment is not yet ready. Once that environment's document scripts are loaded, its execution will begin, with the added effect of starting any applets contained within it.

7.2. Integration with X Mosaic

The iHTML glue to the X Mosaic browser is responsible for transforming iHTML browser function calls into the operations needed for the browser. Its implementation is for the most part straightforward, as many of the functions map directly into functional counterparts within the browser. The browser glue is implemented in three main files. `IHBrNCSA.c` includes the functions that operate directly with the browser, such as HTML markup manipulation and URL retrieval. `IHBrXWin.c` encapsulates the interface to the X Window System, for manipulating the IHWidget, IHPixmap, IHColor, and IHFont types. Finally, the `IHBrUnix.c` file implements low-level Unix access, including path retrieval, directory manipulation, and access to shared libraries.

7.2.1. HTML Parse Trees

The most complicated part of the interface to the browser itself is access to its HTML markup trees. This is primarily because NCSA Mosaic does not represent HTML documents as a true tree—instead, it stores a simple flat list of the tags and text that it found in the document. As such, the browser glue must either convert the iHTML tree manipulation functions into list operations, or transform the parse lists generated by the browser into true trees. In the interest of efficiency, a combination of the two approaches is used.

Figure 7.2 illustrates how this transformation is performed. The iHTML glue adds two new fields to each HTML. One of these points to the previous node in the list, while the other points to the node's parent in the tree that is being created. However, in order to reduce the amount of memory used by each node, the structure does not include pointers for traversing the immediate children of a node (for example, to move from `<HEAD>` to `<BODY>` in Figure 7.2). Instead, the system must follow the next or previous links to find the correct node.

The function to transform the list into a tree moves through the list, filling in the new fields. When it encounters a node with a non-empty content type, it recursively calls itself with the node it found as the new parent whose children it is filling in. In addition, the routine must deal with the complication of missing end tags (particularly common in the case of the `<P>` tag), inserting nodes for them into the tree where appropriate. The pseudocode for this function is shown in Figure 7.3.

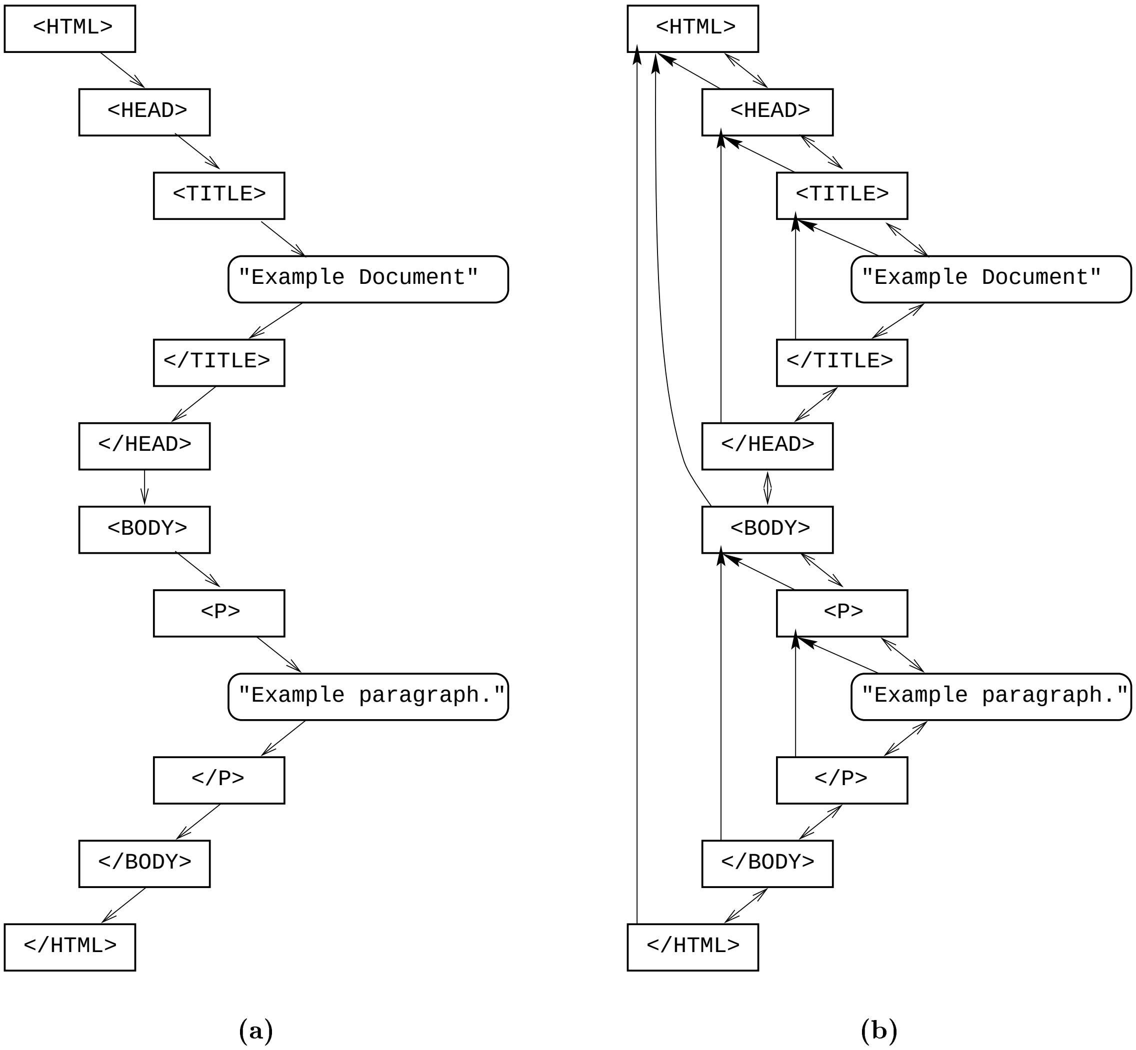


Figure 7.2 Transforming a list of HTML nodes into an actual tree: (a) The original list of HTML nodes created by Mosaic; (b) The addition of "previous" and "parent" pointers to create a tree

```
function make_html_tree(HTMLNode parent, HTMLNode list)

  (* Remember where this branch started. *)
  variable HTMLNode root
  root ← list

  (* Fill in the previous pointer for this node. *)
  if( list ≠ null ): list.previous ← parent

  (* Run through all nodes in this branch. *)
  while( list ≠ null ):

    (* Fill in this node's "parent" and the next node's "previous." *)
    list.parent ← parent;
    if( list.next ≠ null ): list.next.previous ← list;

    (* End this branch of the tree? *)
    if( parent ≠ null ):
        and is_end_node(list)
        and content_type(list.type) ≠ CONTENT_EMPTY ):

      (* Need to insert a missing end node *)
      if( parent.type ≠ list.type ):
        variable HTMLNode node
        node ← alloc_end_node(parent)
        insert node at this position in list.
        list ← node

      return list.next

    (* Move down in tree, or continue in this branch? *)
    if( content_type(list.type) ≠ CONTENT_EMPTY ):
      list ← make_html_tree(list,list.next)
    else
      list ← list.next

  return root
```

Figure 7.3 Algorithm to transform a list of HTML nodes into a true tree structure

7.2.2. Widget and Event Handling

The other major part of the browser glue is the code for managing events and the interface to the X Window System. Much of this is handled in the `IHBrXWin.c` file, which implements a fairly direct mapping from the iHTML's rendering functions to equivalent X11 calls (its IHPixmap functions are build on top of X's Pixmap and GC support, IHColor functions on X's Pixel feature, etc.). These types also need to include additional context information, particularly the X Display where they are located. The glue thus defines a stub structure for each iHTML type, which includes the X resource being managed and other needed state information.

A more complicated part of the code is the IHWidgetRep implementation. This is built on top of a custom Xt widget called `XihGeneric`, which is a simple subclass of the Xt `Constraint` class. The widget encapsulates all the code needed to interface with the rest of the iHTML system. It keeps track of the language environment and language-side object associated with the widget, and provides its own Pixmap that can be used for rendering. In addition, it supplies a translation table that funnels all X widget events into one function, which then transforms them into iHTML IHEvent structures and sends then to the language-side object to be handled.

The implementation of the glue's IHEvent code is itself a troublesome issue, primarily due to the existence of the two event classes. Widget events should be implemented by the X glue, while document events need to be implemented by the browser glue. To solve this, the X glue implements a basic IHEvent that encapsulates the information contained in an X event, along with two functions for getting and setting its attributes, `BRXWin_GetEventAttr()` and `BRXWin_SetEventAttr()`. The browser glue then implements the iHTML event functions, layering its document events on top of the X event interface.

Finally, while widget events are automatically generated and sent by the X widget implementation, the browser glue must explicitly generate document events itself. This is accomplished by inserting code at specific points in the browser where these events should occur (when scrolling the window, printing, highlighting anchors, etc.). The code to do this is one of the primary intrusions into the actual browser implementation that is needed by the iHTML system. The changes are relatively simple—generally a call to `BR_AllocEvent()`, entering data into the event, and then a call to `BR_HandleEvent()`—but are distributed throughout the browser implementation.

7.3. Language Modules

The final major part of the iHTML implementation is that of the language modules themselves. Two example language modules have been implemented, a *python* module and a *text* module. The former encapsulates a Python version 1.3 language interpreter, demonstrating how to write a module for a complete scripting language. The latter, in contrast, simply displays textual files; it serves as an example of a language module that displays a data type (like a traditional plug-in). Sections 7.3.1 and 7.3.2 provide details on the most complicated parts of the language module implementation: managing language environments and code modules, and the language-side widget implementation, respectively.

A language module that implements a true scripting language will need to define the interface between scripts and the underlying iHTML system. There are two major issues—defining how a script exposes its top-level (i.e., "main") interface

```
import _ih_url

class RawURL:

  def __init__(self,doc=None,url=None,**args):

    # Retrieve and stash low-level objects
    if( args.has_key('_URL_') ):
      self._URL_ = args['_URL_']
    else:
      if string.lower(url[0:5]) == 'file:':
        raise SecurityError, "access not allowed to file: URL"

      # This automatically attaches _URL_ to our class instance.
      _ih_url.NewURL(doc._Document_,(url,self))

    self._URL_Name_ = self._URL_.URL
```

Figure 7.4 Initialization implementation of the Python URL class

to the language module, and determining how the script will create and manipulate various iHTML resources such as the IHPixmap, retrieve URLs, etc. The Python module takes care of the first issue through its `ihEmbed` and `ihMain` symbols, and their associated Application and Document classes. For the second, in general, the language module directly exposes the underlying iHTML type as a basic Python type, and then wraps it in a Python class.

This situation is illustrated in Figure 7.4, representing part of Python's URL class implementation. The "`_ih_url`" module is the C interface to the iHTML system's IHRetrieveRequestP type and related functions, encapsulating it as a Python type. It exports one function, "`NewURL()`," which creates an object of this type.

When the RawURL class is instantiated, it checks for one of two conditions. If a "`_URL_`" argument is present, it is an existing URL type with which the class is to interface. If no low-level object is supplied (which corresponds to the common case of the script developer directly instantiating this class), a new low-level URL object is created (the "else" part of the conditional). Here the class first checks to make sure that a "`file:`" URL is not being created, which would be a security hole.[1] If the URL is okay, the code calls the low-level `NewURL()` function to start a new URL request. The first argument is the document to which the URL is being retrieved; here, the code extracts the low-level document object from the document instance that was given to it. The second parameter supplies the name of the URL to retrieve and a handle on the high-level URL instance object. When the low-level URL is created, it attaches itself to the high-level object as "`_URL_`"; this is then used (in the last line) to retrieve the full name of the URL being retrieved.

7.3.1. Script Environment and Protection

One of the central responsibilities of a scripting language module is the management of execution environments and code segments. These are expressed in the iHTML system as the IHLangEnv and IHCodeInfo types, respectively. The former encapsulates the execution environment for an entire document; it is generally used to store information needed by the document script, and other "global" environment information. The latter contains information the module needs to keep about an individual code segment (i.e. the compiled form of an applet or document script).

The language environment used by Python is shown in Figure 7.5. At the top is the "`ih_lang_env_rec`" structure, which is the iHTML public portion of the environment. After that are the fields that are initialized only when the environment

[1] Future versions of the library may transform such URLs into a file request from the user. This would allow scripts to gain safe access to the user's files.

```
struct priv_lang_env_rec {
  struct ih_lang_env_rec env;

  /* Static variables that are made at AllocEnv() time. */
  PyObject* environment; /* The instantiated SafeExec environment */
  PyObject* exec;        /* Function to execute a code block */
  PyObject* eval;        /* Function to evaluate a code block */
  PyObject* install;     /* Function to install module code block */

  /* Dynamic variables that are made at each run. */
  PyObject* doctype;     /* Low-level PyDocument for environment */
  PyObject* docobj;      /* Either the above object,
                              or the user's document object */
  PyObject* module;      /* Module context:
                              __main__ copied from SafeExec */
  PyObject* dictionary;  /* Global dictionary:
                              __main__ dictionary */
  PyObject* requests;    /* List of PyURL objects */
};
```

Figure 7.5 Language environment used by the Python module

is first created. These are used to establish a restricted execution environment that is shared by all code executing in the document, which will be discussed later. Last are the fields that are created each time the language environment begins execution. When this happens, a new "PyDocument" object is created, which provides script access to the browser document just like the URL type discussed above. If the environment has a document script, this is executed and its resulting instance used as the high-level handle on the document; otherwise, the PyDocument object itself is used as this handle. A module named "`__main__`" is created in the restricted execution environment for the document script to execute, and a handle on that module is stored in the language environment. Finally, the environment also tracks active URL transactions, so that it can delete them when terminating.

```
struct python_code {
  /* Created only when first allocated */
  PyCodeObject* program;   /* The compiled code */
  PyObject* module;        /* The module it created */

  /* Embedded objects: created each time code execution started. */
  PyObject* arg_dict;         /* Dict. of args for embedded obj */
  PyInstanceObject* embedded; /* Current class attached to widget */
};
```

Figure 7.6 Code context used by the Python module

Script security is ensured through the "`safeexec.py`" module. This is based on Python's own "`rexec.py`" module, providing a controlled execution environment. In addition, it implements the "`__export__`" and "`__public__`" symbols, using them to determine which parts of a module in this environment should be visible to other modules. One safe execution environment is created for each document, and all codes executing inside that document—both applets and document scripts—share this environment. This allows the various programs to have a moderate amount of interaction with each other, but protects them from code executing in separate documents.

The final piece to script execution is Python's IHCodeInfo structure, shown in Figure 7.6. This is used to hold information about a single piece of code. It is allocated by the language module in its `AllocModule()` and `AllocEmbedded()` functions (for document scripts and applets, respectively). When created, a language environment already exists for the document; the system creates a new module for the code to execute in this environment, and compiles it there. In addition, it stores information about the program arguments (`<PARAM>` tags) and language-side widget object (for applets) for each program run.

7.3.2. Widget Interface

In language modules that implement true scripting languages, the implementation of their language-side widget objects is very similar to other interfaces to the iHTML system. That is, a bridge is created between the high-level scripting language and iHTML's widget architecture. The Python module does this just as described for other interfaces, implementing the widget as a low-level Python type and wrapping it in a high-level class. The only complication is that browser-side widgets themselves do not define any behavior, so the language must implement its own strategies for such things as widget layout.

Language modules that implement data types usually work a different way. Figure 7.7 illustrates this, showing the definition of the IHCodeInfo and IHWidgetObj structures that are used by the text language module. Because such modules are not working as a bridge to a scripting language, they must implement all the actual behavior they are to exhibit. Generally, these two structures are very closely linked. The IHCodeInfo structure represents the global information for a single embedded object, and the IHWidgetObj structure is used to display that object. The language module may implement a slightly more sophisticated IHWidgetObj structure, if it is going to make use of a widget instance hierarchy; this structure definition is shared among all widgets in the hierarchy.

The text language module uses its IHCodeInfo to store the information it retrieved from the applet arguments about the size its embedded object should be, along with the dynamic widget data that is created when it begins "execution." At that point, it is given a browser-side widget to render in, so it allocates one of its own IHWidgetObj to interact with the other widget. The IHWidgetObj is simply used to interface with a raw X widget—it allows that X widget to perform its own layout,

```
struct ih_int_lang_code_rec {

  /* Created when code block is first allocated. */
  int width, height;        /* The requested applet size. */

  /* Created each time execution is started. */
  IHWidgetObj obj;          /* Top-level widget interface. */
  Widget viewport;          /* The scrollable area. */
  Widget text;              /* The text being displayed. */
};

struct ih_int_widget_obj_rec {
  IHWidgetRep rep;          /* Browser widget associated with. */
  IHWidgetRaw raw;          /* Raw OS-level widget object.  For the
                               Xt interface, this is a Widget obj. */
  Dimension width, height; /* Preferred dimensions. */

  Widget child;             /* X-level child widget. */
};
```

Figure 7.7 The code and widget structures used by the text language module

and then transmits that information to the browser-side widget. Thus the language module can simply create a few X widgets in which the text will be displayed, and connect them to its IHWidgetObj.

8. CONCLUSIONS

There are currently three classes of client-side services that are in widespread use in Web browsers. **Plug-ins** are compiled code modules stored on the same machine as the browser, which work in cooperation with that browser to implement new capabilities. **Applets** are architecture-independent pieces of code that a browser retrieves from the network and executes on its local machine in order to implement new features in a restricted area of the document. **Document Scripts** are architecture-independent pieces of code that work with the browser to control the display of, and user interaction with, an entire document. Currently, there are many competing implementations of the three service classes, which were never designed to work together. One result is a pronounced lack of interoperability and consistency among the various client-side services.

Interactive HTML is a system that unifies client-side services. It defines an extended plug-in interface that allows plug-ins to be written not just for traditional data types but also for scripting languages that implement applets and document scripts. The iHTML system is structured in three main components. The **iHTML Library** is the only one actually implemented by the system, and serves as the central regulator to control the interaction between a Web browser and the back-end language modules. A **Language Module** is the extended plug-in interface, implementing one or more MIME content types and instructing the rest of the system how to display or otherwise handle that data. The **Browser** is the traditional Web browser, to which the system defines a standard interface but otherwise is treated as a black box.

The iHTML design also unifies the treatment of “data” and “programs,” allowing the system to handle any type of file it encounters, without concern for what it actually is. As a result, applets and document scripts are decoupled from the browser itself. There is no need for the code handling them to be compiled directly into the browser, so they require system resources only when actually in use. In addition, a browser can easily support arbitrary scripting languages; the scripting languages can be updated, or new ones added, without any changes to the browser executable. Language modules can also be shared among all browsers implementing the iHTML architecture.

To date, iHTML has been incorporated into NCSA's X Mosaic version 2.7 browser and two language modules have been written. The first is the *python* language module, which encapsulates a full Python 1.3 interpreter. It is an example of a language module providing complete scripting services (supporting both applets and document scripts), as it implements a full Python scripting interface to the rest of the iHTML system (including widgets and HTML parse trees). The second language module is *text*, which is an example of a simple module handling a traditional data file. It simply displays any ASCII text as a scrollable view, embedded in an HTML document.

8.1. Existing Limitations

The iHTML system currently is oriented toward browsers that don't use progressive display (i.e., that retrieve all the data for a document before displaying it to the user). Most modern browsers, however, support progressive display, and using that feature increases user productivity. The primary limitation the iHTML system has with progressive display is that the system can't accurately display a document until the script has been executed and has had a chance to manipulate the document's HTML parse tree. Therefore, the browser must wait until the entire script and document both have been retrieved and the script executed, before displaying anything to the user.

In addition, while the HTML parse tree manipulations implemented by iHTML provide the same functionality as the existing practice of embedding `<SCRIPT>` tags inside a document (see Section 2.3.2), there are many situations where the `<SCRIPT>` syntax is much more convenient. The problem is that the binding from an HTML document to its document script has become too weak, necessitating that some scripts become too dependent on the documents themselves. For example, a document script that inserts the result of a database query into its document must know enough about that document to be able to determine where in its HTML parse tree the new data should be inserted.

One solution to the weak binding between HTML document and script may be the introduction of a `<CALL>` tag, shown in Figure 8.1. When the `<CALL>` tag is encountered in a document, it would result in a function being executed in the document script. The function could return an HTML parse tree, which would be inserted into the document at the place the `<CALL>` tag appeared. Figure 8.2 shows an example of how such a tag might be used to insert the result of a database query.

The `<CALL>` tag could also assist iHTML in progressive document display. A system implementing this tag could retrieve a document script asynchronously while progressively displaying the document text. When it encountered a `<CALL>` tag, the system would wait for the document script to finish being retrieved, and start execution before performing the call. Such a design would allow script execution to be synchronized with document display.

Another limitation of iHTML is in the area of forms and graphical user interfaces. The current iHTML system does not provide access to the browser's internal form widgets, such as pop-up menus and text entry fields, so applets can't

```
<!ELEMENT call - - (param | %bodytext)*>
<!ATTLIST call
  function CDATA   #REQUIRED  -- name of function to call --
  script   %URL    #IMPLIED   -- name of target document script --
  >

<!ELEMENT param - O EMPTY -- named property value -->
<!ATTLIST param
  name      CDATA             #REQUIRED  -- property name --
  value     CDATA             #IMPLIED   -- property value --
  >
```

Figure 8.1 DTD of the proposed `<CALL>` tag

```
<P>Query Results:</P>
<BLOCKQUOTE>
  <CALL FUNCTION="insert_query">
    <PARAM NAME="query" VALUE="+llama +pet">
  </CALL>
</BLOCKQUOTE>
```

Figure 8.2 Example use of the proposed `<CALL>` tag

create custom user interfaces with controls matching those used by the browser. A solution to this problem may be to allow different "types" of widgets to be created, so that `BR_AllocWidget()` can create either a basic widget, or one of the higher-level control widgets used by the browser. Similarly, there is no way to create windows and dialog boxes through a browser running under iHTML. The definition of an IHWindow type and functions for creating and manipulating windows would greatly expand the user interface capabilities of iHTML.

In addition, the iHTML system's user interface functionality provides no way for scripts to access fonts (other than the default font). Typically, fonts are accessed using attributes such as name (e.g., "times"), style ("bold"), size ("12 point"), etc. While this type of low-level interface will continue to be needed, iHTML's access to the HTML parse tree presents the opportunity for an interesting high-level interface. Consider the case where a script should use the same font as the browser uses for its `<H1>` text. With the traditional mechanism for accessing fonts, this is difficult or impossible, as the font associated with a particular tag depends on the browser implementation, user preference settings, and possibly many other variables. Nevertheless, it might be possible to request a font as a set of HTML tags. For example, a call to `BR_GetFont("<H1>")` could return the font used in the `<H1>` tag. Or consider a more complicated example: `BR_GetFont("<P><SMALL><EM>")` could result in the font used to render small emphasized text in the body of the document.

There also needs to be more communication between scripts written in different languages. The primary mechanism for communication is the IHEvent structure, which unfortunately only supports attribute values and event types that are explicitly defined by iHTML. To address this limitation, the Python language module extends its own event class to allow custom attributes, as well as class and code values. Such extensions, however, are lost when events move through the underlying iHTML system to other language modules. Adding Python's event extensions to the library itself would greatly increase the amount of communication that can occur between language modules. Another solution may be to incorporate a full inter-language communication system, such as the Inter-Language Unification System (ILU) [21].

The current interface for retrieving URLs is very limited, allowing URLs to be retrieved and accessed only as raw binary data. For example, there is no way to request a URL for a GIF picture, which should be returned as an IHPixmap object that can be used by an applet to draw its display. The solution here may be to add an additional parameter to `BR_RetrieveURL()` that specifies the type of data that is expected, and to generalize the `IHBufferP` interface to be able to hold higher-level types (such as pixmaps). In addition, while the current system completely blocks the `file:` URL due to security concerns, there are many occasions when an applet needs access to files on the client machine. Transforming `file:` URL requests to file requesters shown to the browser user could provide such access in a safe way.

Finally, the iHTML system needs a more sophisticated view of script execution state. Currently, a script is either running or stopped; when it is restarted after being stopped, its state is reset completely. Since scripts are stopped in a document that currently is not being viewed, moving back through previously viewed documents results in their scripts being started again. Many times, however, a script needs to begin execution from the place it was when the document was hidden. For example, an applet for drawing a picture needs to be able to remember what the user had drawn previously. This need goes beyond execution state within a given session. It would also be useful to allow scripts to store state and preference information between browser sessions. Such storage could be accomplished by introducing a mechanism allowing scripts to write a certain amount of state information to disk, associated with the particular URL from which the script was retrieved; it could later be read back by that script.

8.2. Lessons Learned

As the Web continues to develop, it seems likely that a wider variety of scripting languages will come into use. Providing enough access to the browser so that scripting languages—particularly those used for document scripts—can be implemented as plug-ins like traditional data types, yet remain well-integrated with the browser, may help greatly in addressing the growing number of languages. The combination of data and program handling itself presents a number of advantages. A browser no longer needs to be concerned about the type of data embedded in an HTML document. Both "program" and traditional "data" types are handled in the same way by the browser. In addition, this unification of programs and data may present the opportunity for creating new kinds of plug-ins that exhibit characteristics of each.

A standing issue is the syntax used to associate a document script with an HTML document. Currently, most browsers make use of either the `<SCRIPT>` tag or a `<LINK>` tag that appears in the `<HEAD>` of a document (in fact, the iHTML system originally used the `<LINK>` tag itself). However, with the unity between document scripts and applets, it seems to make more sense to use the same basic HTML syntax as well, hence iHTML's use of the `<OBJECT>` tag and `CONTEXT` attribute. This new syntax actually serves to simplify the underlying system, while at the same time making document scripts more powerful. Using this syntax allows for a more sophisticated specification of a document script's URL and content type, as well as allowing arguments to be passed to the script.

Another iHTML concept that holds many possibilities is opening up access to a browser's internal HTML parse tree. The most obvious advantage here is the fact that document scripts can now examine as well as construct parts of an HTML document. Such capabilities help tremendously in allowing general-purpose scripts to be written. More interesting is the fact that scripts can now dynamically *change* a document. This opens many possibilities, such as implementing hyperlinks that retrieve another document and embed it into the current document in place of the link, being able to hide part of the text in a document, automatically generating an index of terms in a document, etc. Finally, as style sheets come into more widespread use, a document's HTML parse tree could also provide access to the style information associated with each node, allowing such things as document scripts that dynamically change the formatting of a document or even implement their own custom style language.

One important use of iHTML's document scripts is in the collection of Web usage information. Traditionally, only simple statistics, such the HTTP hit count on each document, have been available. Some proposed approaches may provide better information, particularly about the path individual users take through an entire HTTP server, but none of these approaches supply information at a finer grain than the document level. By associating a document with an iHTML script that watches the user interact with the document, it is possible to collect more detailed information, such as movement through the document, use of hyperlinks, manipulation of forms, etc. Such data is important because it reveals how much of a page users actually view and whether they read the page in the order that was intended.

Finally, the iHTML system makes some attempt to help increase the source-level portability of language modules across computer platforms. The most important way it does this is by defining the IHWidgetRep type, which provides an abstract interface to the underlying operating system's user interface drawing and event handling services. Unfortunately, there is not enough implemented by iHTML for this abstraction to be really effective. At the least, a method of accessing the browser's form widgets and creating and managing windows (as described in Section 8.1) must also be provided. Even if a complete abstract GUI is defined by the system, it would still need to provide access to the underlying operating system objects (through the IHWidgetRaw type) for language modules that need to make use of special features of a particular platform.

IHTML integrates traditional plug-ins, applets, and document scripts into one uniform and consistent system. In doing so, iHTML both increases browser flexibility and makes it possible to reuse language and data handlers across multiple browsers.

BIBLIOGRAPHY

[1] C. S. Ang, D. C. Martin, and M. D. Doyle. Integrated Control of Distributed Volume Visualization through the World-Wide-Web. In *Proceedings of the IEEE Visualization 1994 Conference*, October 1994. Also available: `http://www.eolas.com/papers/Papers/IEEE-VIS-94/`.

[2] Steve Ball. New Approaches to Custom WWW Interfaces. In *Proceedings of the 1995 AUUG/Asia-Pacific WWW Conference*, Sydney, Australia, September 1995. Also available: `http://surfit.anu.edu.au/SurfIt/papers/apwww/apwww.html`.

[3] T. Berners-Lee and D. Connolly. Hypertext Markup Language - 2.0 [Online]. RFC 1866, Internet Engineering Task Force (IETF), November 1996. Available: `ftp://ds.internic.net/rfc/rfc1866.txt`.

[4] T. Berners-Lee and R. Fielding. Hypertext Transfer Protocol – HTTP/1.0 [Online]. RFC 1945, Internet Engineering Task Force (IETF), May 1996. Available: `ftp://ds.internic.net/rfc/rfc1945.txt`.

[5] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, Internet Engineering Task Force (IETF), December 1994. Also available: `ftp://ds.internic.net/rfc/rfc1738.txt`.

[6] Krishna Bharat and Luca Cardelli. Distributed Application in a Hypermedia Setting [Online]. Technical Report, Georgia Institute of Technology and DEC Systems Research Center, 1996. Available: `http://www.cc.gatech.edu/gvu/people/Phd/Krishna/IWHD.html`.

[7] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) [Online]. RFC 1341, Internet Engineering Task Force (IETF), June 1992. Available: `ftp://ds.internic.net/rfc/rfc1341.txt`.

[8] François Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th World Wide Web Conference*, Paris, France, May 1996. Also available: `http://www5conf.inria.fr/fich_html/papers/P41/Overview.html`.

[9] Dan Connolly. Proposals for Gathering Consumer Demographics [Online]. Tech Note, World Wide Web Consortium, November 6 1995. Available: `http://www.w3.org/pub/WWW/Demographics/Proposals.html`.

[10] Sean Dorward, Rob Pike, and Phil Winterbottom. Inferno: la Commedia Interattiva [Online]. Technical introduction, Lucent Technologies, Murray Hill, NJ, July 1996. Available: `http://inferno.bell-labs.com/inferno/infernosum.html`.

[11] M.D. Doyle, C.S. Ang, and D.C. Martin. Proposing a Standard Web API. *Dr. Dobbs Journal*, pp. 18–26, February 1996.

[12] Joe English. Notes on SGML and HTML [Online]. Commentary, October 1995. Available: `http://www.art.com/~joe/sgml/`.

[13] Joe English. Cougar DTD: Do not use CDATA declared content for SCRIPT. WWW-HTML [Online]. July 23 1996. Available Email: `http://www.art.com/~joe/sgml/cougar-cdata.txt`.

[14] Michael Franz. Code-Generation On-the-Fly: A Key to Portable Software. Dissertation, ETH Zrich, Switzerland, February 1994.

[15] Michael Franz and Thomas Kistler. Juice Homepage [Online]. Project home page, June 17 1996. Available: `http://www.ics.uci.edu/~juice/intro.html`.

[16] Geri Gay and Joan Mazur. The Utility of Computer Tracking Tools for User-Centered Design. *Educational Technology*, 33(4):45–59, April 1993.

[17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996. Also available: `http://java.sun.com/doc/api_documentation.html`.

[18] James Gosling and Henry McGilton. The Java Language Environment: A White Paper [Online]. White Paper, Sun Microsystems, October 1995. Available: `http://java.sun.com/doc/language_environment/`.

[19] James Gosling, Frank Yellin, and the Java Team. *The Java Application Programming Interface*. Addison-Wesley, Reading, MA, 1996. Also available: `http://java.sun.com/doc/api_documentation.html`.

[20] International Standards Organization. *ISO 8879: Information processing—Text and office systems–Standard Generalized Markup Language (SGML)*. ISO, Geneva, Switzerland, October 1 1986.

[21] Bill Janssen and Mike Spreitzer. *ILU Reference Manual* [Online], version 2.0. Xerox Corporation, July 1996. Available: `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[22] Charlie Kindel, Lou Montulli, Eric Sink, Wayne Gramlich, Jonathan Hirschman, Tim Berners-Lee, and Dan Connolly. Inserting objects into HTML [Online]. Working Draft, World Wide Web Consortium, April 22 1996. Available: `http://www.w3.org/pub/WWW/TR/WD-object.html`.

[23] Douglas Kramer. The Java Platform [Online]. White Paper, Sun Microsystems, May 1996. Available: `http://java.sun.com/doc/whitePaper.Platform/CreditsPage.doc.html`.

[24] David M. Kristol and Lou Montulli. HTTP State Management Mechanism. *World Wide Web Journal*, 1(4):199–211, Fall 1996.

[25] Jacob Levy. A Tcl/Tk Netscape Plugin [Online]. Early draft, May 24 1996. Available: `http://www.smli.com/research/tcl/plugin/paper.html`.

[26] Tim Lindholm and Frank Yellin. *The Java(tm) Virtual Machine Specification.* Addison-Wesley, Reading, MA, 1996. Also available: `http://java.sun.com/doc/language_vm_specification.html`.

[27] Mark Lutz. *Programming Python.* O'Reilly and Associates, October 1996.

[28] D. C. Martin, C. S. Ang, M.E. Salomon, and M.D. Doyle. Libraries in the Information Age. Technical Report, UCSF Library and Center for Knowledge Management, November 1993. Also available: `http://www.eolas.com/papers/Papers/LibraryInformationAge/`.

[29] Microsoft Corporation. *Creating Platforms for Innovative Internet Software* [Online], June 25 1996. Available: `http://www.microsoft.com/workshop/prod/strat/intover-f.htm`.

[30] Microsoft Corporation. *Microsoft Site Builder Workshop* [Online], May 28 1996. Available: `http://www.microsoft.com/workshop/author/cpad/tutorial-f.htm`.

[31] Microsoft Corporation. *What is ActiveX?* [Online], June 6 1996. Available: `http://www.microsoft.com/intdev/sdk/actlogo-f.htm`.

[32] D. Morris, C.J. Theaker, R. Phillips, and W. Love. Human-Computer Interface Recording. *Computer Journal*, 31(5):437–44, October 1988.

[33] National Center for Supercomputing Applications. *NCSA Mosaic Common Client Interface* [Online], version 1.1. University of Illinois, March 31 1995. Available: `http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-spec.html`.

[34] A. S. Neal and R. M. Simons. Playback: A method for evaluating the usability of software and its documentation. *IBM Systems Journal*, 23(1):82–96, 1984.

[35] Alan S. Neal. Measuring software usability—a methodology and case studies. In *Proceedings of the 1988 IEEE International Conference on Systems, Man, and*

Cybernetics, Vol. 31, pp. 189–92, Int. Acad. Publishers, Beijing, China, August 1988.

[36] Netscape Communications Corporation. *The Java Runtime Interface* [Online], version 0.6, June 28 1996. Available: `http://home.netscape.com/eng/jri/index.html`.

[37] Netscape Communications Corporation. *The JavaScript Guide* [Online], version 3.0, 1996. Available: `http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html`.

[38] Netscape Communications Corporation. *LiveConnecting Plug-ins with Java* [Online], version 3.0, 1996. Available: `http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/pjava.htm`.

[39] Netscape Communications Corporation. *Plug-in Guide* [Online], version 3.0, 1996. Available: `http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/pguide.html`.

[40] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[41] John K. Ousterhout. The Relationship between Tcl/Tk and Java [Online]. Technical Report, Sun Microsystems Laboratories, April 7 1996. Available: `http://www.smli.com/research/tcl/java.html`.

[42] Dave Raggett. HTML 3.0 Document Type Definition [Online]. Technical Report, World Wide Web Consortium, March 24 1995. Available: `http://www.w3.org/pub/WWW/MarkUp/html3/html3.dtd`. Expired draft.

[43] Dave Raggett. HTML 3.2 Document Type Definition [Online]. Technical Report, World Wide Web Consortium, August 21 1996. Available: `http://www.w3.org/pub/WWW/MarkUp/Wilbur/HTML32.dtd`.

[44] Antonio C. Siochi and Roger W. Ehrich. Computer Analysis of User Interfaces Based on Repetition in Transcripts of User Sessions. *ACM Transactions on Information Systems*, 9(4):309–335, October 1991.

[45] John B. Smith, Dana Kay Smith, and Eileen Kupstas. Automated Protocol Analysis. *Human-Computer Interaction*, 8:101–145, 1993.

[46] C. M. Sperberg-McQueen and Lou Barnard. A Gentle Introduction to SGML. Technical Report, Text Encoding Initiative, May 28 1993. Also available: `http://www.ebt.com/usrbooks/teip3/1336`.

[47] Paul Thistlewaite and Steve Ball. Active FORMs. In *Proceedings of the 5th World Wide Web Conference*, Vol. 5, Paris France, May 1996. Also available: `http://www5conf.inria.fr/fich_html/papers/P40/Overview.html`.

[48] Deborah Trumbull, Geri Gay, and Joan Mazur. Students' Actual and Perceived User of Navigational and Guidance Tools in a Hypermedia Program. *Journal of Research on Computing in Education*, 24(3):315–28, 1992.

[49] Guido van Rossum. Grail – The Browser For The Rest Of Us [Online]. Technical Report, Corporation for National Research Initiatives, Reston, VA, May 30 1996. Available: `http://monty.cnri.reston.va.us/grail/info/papers/restofus.html`. Early draft.

[50] Noriko Yamagishi and Motei Azuma. Experiments on human-computer interaction evaluation. In G. Salvendy, editor, *Cognitive Engineering in the Design of Human-Computer Interaction and Expert Systems*, Vol. 2, pp. 167–74, Elsevier Science, Amsterdam, Netherlands, 1987.

APPENDICES

APPENDIX A. Python Script Developer Manual

APPENDIX B. Language and Browser Developer Manual