APPLICATION DEVELOPMENT WITH HYPERTEXT MARKUP LANGUAGE (HTML)

by

Robert Joel Dunn

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1995

Approved by:

_____ Advisor

_____ Reader

_____ Reader

© 1995
Robert Joel Dunn
ALL RIGHTS RESERVED

ROBERT JOEL DUNN. Application Development with Hypertext Markup Language (Under the direction of Stephen F. Weiss)

ABSTRACT

This paper examines the use of Hypertext Markup Language (HTML) in the creation of distributed heterogeneous application systems. First, it presents a basic description of the World Wide Web (WWW) and its standard client/server interaction. Then, it briefly covers HTML and its relationship to client processes. The discussion then progresses to language elements of HTML that permit the simple request-response model to be expanded into application systems. After these building blocks of application development are exposed, problems with this model as an application development environment and some potential solutions to these problems are noted. Since tool suites are highly important to successful application development, this paper provides a discussion of some popular tools, such as editors and syntax verification devices. To illustrate some of the issues in project development, a survey of some existing network applications is presented.

Chapter 1 - Introduction 1
Chapter 2 - Background 5
Chapter 3 - History and Concepts of the World Wide Web 11
3.1 - History 11
3.2 - Basic Concepts 13
Chapter 4 - World Wide Web Browsers 21
Chapter 5 - What is an HTML Application? 24
Chapter 6 - Application Development 37
6.1 - Constructs in Application Development 37
6.2 - How is an application accessed? 44
Chapter 7 - Browser Differences 45
7.1 - Application Appearance on Diverse Clients 45
7.2 - Feature Set of Target Browsers 47
Chapter 8 - Data Access Tools 48
Chapter 9 - Further Problems in Application Development 56
9.1 - Language Issues 56
9.2 - Statefulness and Recovery 58
9.3 - Security 62
9.4 - Concurrency 70
9.5 - Development Tools 74
Chapter 10 - Survey of Existing Applications 77
10.1 - Research 77
10.2 - Commercial Applications 79
10.3 - Games 83
10.4 - Computer-Based Training 85
Chapter 11 - Conclusion 87

Chapter 1 - Introduction

This paper will explore the use of Hypertext Markup Language (HTML) in the creation of distributed heterogeneous application systems. First, it will present a basic description of the World Wide Web (WWW) and its standard client/server interaction. Then, it will briefly cover HTML and its relationship to client processes. The discussion will then progress to language elements of HTML that permit the simple request-response model to be expanded into application systems. After these building blocks of application development are exposed, problems with this model as an application development environment and some potential solutions to these problems will be noted. Since tool suites are highly important to successful application development, this paper will provide a discussion of some popular tools, such as editors and syntax verification devices. To illustrate some of the issues in project development, a survey of some existing network applications will be presented.

The material covered here assumes a working knowledge of UNIX™, TCP/IP, and client/server computing. A good discussion of TCP/IP and socket-based client/server systems can be found in Comer and Stevens, *Internetworking with TCP/IP volume III*.[1] For a good, brief overview of UNIX, see Quarterman, et. al., "4.2BSD and 4.3BSD as Examples of the UNIX System" [Quarterman, 1985]. Additionally, a working familiarity (as a user) with the World Wide Web and HTML is advantageous, but not mandatory. The sufficiently informed reader may skip the introductory chapters (*Background*, *History and Concepts of the World Wide Web,* and *World Wide Web Browsers*) and begin with the material presented in Chapter 5, entitled *What is an HTML Application?*.

When the reader has finished this document, he will have been exposed to the use of HTML and the World Wide Web as an application development environment. He should be able to take the examples presented herein and construct simple application systems that permit experimentation with the

[1] COMER, DOUGLAS, STEVENS, DAVID "Internetworking with TCP/IP, Volume III", Prentice-Hall, Inc., *ISBN 0-13-474222-2*, 1993

development facilities available. More importantly, the reader should be aware of both the problems inherent in the current model and the potential for creating distributed application systems.

The model of presentation in this paper will be one of incremental detail. In Chapter 2 (*Background)* , which may be skipped by the knowledgeable reader, many basic terms and ideas will be presented in an introductory fashion with limited supporting detail. As the paper progresses and the understanding of the reader increases, these topics will resurface with increasing detail and explanation.

To begin the discussion and to understand the concepts of application development using World Wide Web tools, it is first necessary to establish some basic definitions as they apply to the Web. A reader seeking additional comparative background for the Web and other Internet information navigation tools, as mentioned below, may wish to consult Schwartz, et. al., "A Comparison of Internet Resource Discovery Techniques" [Schwartz 1992].

Browser - A *web client*, the workstation program that interacts via Hypertext Transfer Protocol (HTTP) with a Web server. This program is the user's window into the World Wide Web. The term *browser* was coined since these programs allow the workstation user to quickly skim from one resource to another, much as a reader might quickly flip through the pages of a book, stopping to look at whatever information happened to catch his eye. The reader may safely think of this program as the client in a client/server model. The browser, using HTTP, sends a command string to the server, where it is parsed and processed. The server encodes the results using Hypertext Markup Language (HTML) and again uses HTTP to send the information back to the browser.

CERN - European High Energy Physics Research Lab, the institution whose staff wrote the first Web server and client as a research and information retrieval aid.

CGI - Common Gateway Interface; a standard for executing operating system scripts[2] under the auspices of a HTTP server. When a server receives references to a Uniform Resource Locator (URL) in a specific directory, it is interpreted as host system executable. These operating system scripts perform some function and then dynamically generate HTML to instruct the client in how to format the output. This greatly expands the functionality of the World Wide Web model, allowing system resources (accessible via the scripting language) to be accessed from Web pages.

Gopher - An Internet navigator, designed around a tree-structured directory. Gopher was developed as a simple tool to help organize information (text or graphics) on a particular server and to provide a simple model to retrieve information from other Internet-connected Gopher servers. It is menu-based, reading a tree-structured directory to create local and remote references. Gopher servers may access Wide Area Information Server (WAIS) databases if the server has been compiled with the WAIS libraries. Recent enhancements to this TCP/IP - based service attempt to provide hypertext-like features in a simpler model than the World Wide Web.

HTML - Hypertext Markup Language. This page definition markup language is the basis for both static Web pages and for interactive applications. It defines style and function, specifying text or object type and whether information is local or is on a remote system. HTML is composed of directives coded in ASCII text, imbedded in the document, and usually takes the form "*<directive>target text</directive>*."

HTTP - Hypertext Transport Protocol; a protocol that builds on reliable stream services provided by TCP/IP to carry information between browsers and servers. Just as Telnet and FTP are well-known services in the TCP/IP protocol suite, HTTP has become another such well-known service. Telnet typically operates at TCP port 23, for example, and HTTP is usually serviced at port 80. The protocol specification provides a definition for the stream of octets that arrive at the relevant port.

---

[2]On a UNIX system, these scripts may be Bourne shell, C shell, PERL, or any language supported on the local system. Additionally, they may in fact be compiled programs in a language such as C. Once a HTTP server passes control to a CGI script, the script may do virtually anything! This is obviously a powerful feature, but is certainly one that must be properly managed for system integrity and security.

NCSA - National Center for Supercomputing Applications; this organization has been responsible for much of the popularization of the World Wide Web in North America through creation of the *Mosaic* family of browsers and their own family of servers.

URL - Uniform Resource Locator, a draft standard for specifying an object on the Internet. This specification tells an Internet client what type of object (text, graphics, script, terminal session, etc.) is being referenced, and also where it may be found. The form of a URL is as follows:

resource_type://host_name (or number) :port (optional)/parameters (such as file path)

A typical URL for a Web object might be "http://www.adp.unc.edu/", where this specifies the *http* protocol, the host *www.adp.unc.edu*, and a default port and default path (parameter list). Similarly, the URL "gopher://bullhead.adp.unc.edu:70/1/" indicates the *gopher* protocol, the host *bullhead.adp.unc.edu*, port 70, and the main menu, "*1*".

WAIS - Wide Area Information Server. This tool was designed to index and retrieve information from archives of diverse information. WAIS is another TCP/IP - based service, with its own servers and clients. Direct support for its free-form queries is built into certain Web browsers. Other browsers must use gateways (see the definition of CGI above), or use a Gopher server generated with WAIS support in order to access WAIS-indexed information.

Chapter 2 - Background

HTML[3] is the page definition language that has been popularized by the explosive growth of the World Wide Web, a TCP/IP - based Internet service. Thousands of servers around the world provide data and hypertext references on a wide variety of subjects. The World Wide Web links the popular information retrieval tools Gopher and WAIS, linear text and hypertext, ASCII files and multimedia to create a rich body of information. The Web has been the subject of much recent publicity, and attention has been primarily focused on multimedia presentation of information. The popular view of the Web is as an electronic publishing model. However, this is just one aspect of its capabilities. The Web can combine the traditional data entry/inquiry data processing world with the platform-independent graphical presentation powers of its browsers. The clients and servers of the World Wide Web interoperate without regard for their hardware or software platforms. Analogous to the way that the ubiquitous Kermit[4] file transfer protocol makes it possible to transfer files between virtually any two computers connected with simple telephone links, the HTML pages of the World Wide Web may be read by any compatible client regardless of architecture. Since it is possible through the *forms* feature of the HTML language to actually create application systems, achieving the goal of creating distributed platform-independent applications may well be possible.

The canonical definition of HTML is contained in a *Document Type Definition* (DTD) based on ISO Standard Generalized Markup Language (SGML)[5]. This HTML-DTD is used to define the syntax allowed by editors and supported by browsers. HTML is also an evolving standard, and as we will see later, is actually composed of definitions at various levels of defacto support.

---

[3]http://info.cern.ch/hypertext/WWW/MarkUp/Intro.html
[4]http://www.cc.columbia.edu/kermit/
[5]ISO 8879:1986 Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML), and http://info.cern.ch/hypertext/WWW/MarkUp/SGML.html

Let's take a brief look at some components of the HTML language and an actual example. First, most HTML pages have a title. This is coded "*<TITLE> here is a title </TITLE>*", causing the material between the directives (or tags, as they are sometimes called) to be interpreted by the browser as the title of the page and it may be displayed in the title bar of a browser window, for instance. There are several levels of headings available and each is distinct so that a browser can render them in different fonts or sizes. A first level heading is "*<H1> here is a heading </H1>*". Paragraphs are framed with the construct "*<P> paragraph text </P>*". The paragraph directive is interesting in that the syntax actually permits the end mark "*</P>*" to be omitted, and in practice this is usually the case. A simple but completely valid example of an HTML page is shown below in figure 2.1. Note that as we mentioned above, the HTML author could omit the last line, "*</P>*" and not change the appearance or validity of the page.

Figure 2.1

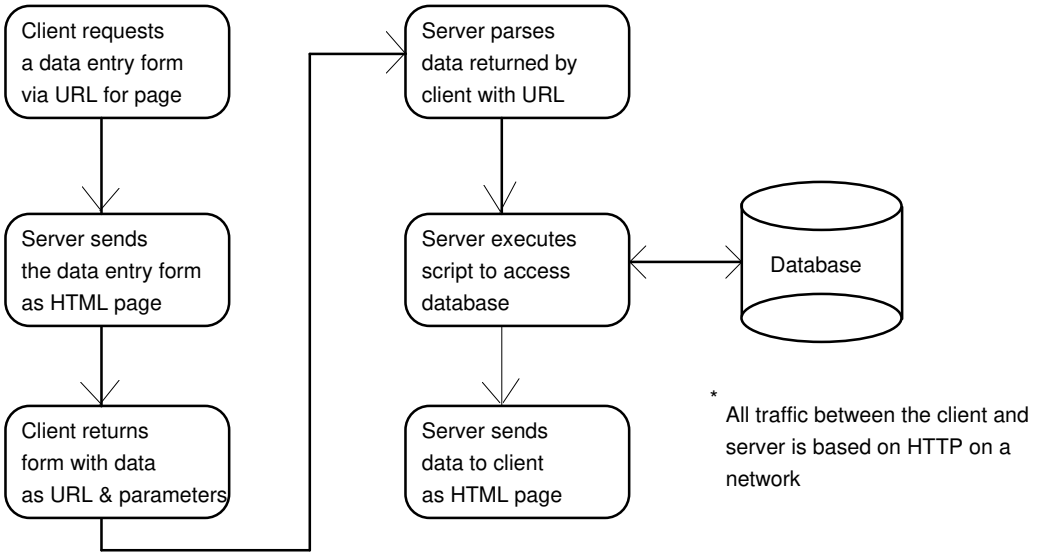
```
<TITLE>This is the title of my page</TITLE>
<H1>This is a heading.  It will be emphasized and will appear with a large font</H1>
<P>
This the body of the page.  It does
not matter if the text lines are long or short, since the browser will not insert a line feed simply because the
source has wrapped to a second line.  Another paragraph directive or similar directive is required to cause text
to flow to the next line.
</P>
```

This is a very basic introduction to HTML. Further commonly used tags and references to more extensive documentation will be covered in Chapter 5. The forms necessary for application development are logically bounded by the directive "*<FORM> form information here </FORM>*". Additionally, there are many specialized tags associated with and contained within forms, and these will be discussed in detail in Section 6.1.

In addition to its duties as a page definition language, HTML is also the programming language of the World Wide Web, although it must invoke scripts in other languages for system and database accesses. Pioneering work in HTML application development has been done at a number of sites, creating HTML pages that access SQL engines and other standard or non-standard databases. Interactive games have been created which utilize the open connectivity of the Web to encourage players to participate. Rudimentary commercial applications are appearing with increasing frequency. A quick perusal of the *What's New* page

at NCSA[6] will give examples. However, there are still many obstacles in the way of wider acceptance of this model. What follows in this chapter is an introduction to some of the problems that application developers using this model must consider. Each of these issues is discussed in detail in later sections of this paper. To motivate this discussion, let's look at a conceptual diagram of a generalized World Wide Web application which accesses a database.

Figure 2.2



Perhaps the two most obvious problems for application development with HTML are the closely related issues of session authentication and transport security. Since the Internet is an open, loosely controlled network, data that is entrusted to the net is open to scrutiny as it passes through many machines from its source to destination. Given the nature of the Internet, few people would today willingly enter sensitive personal or financial information. Work is underway at various commercial and research organizations that will address some of these concerns through encryption and digital signatures.[7]

Other important deficiencies in this model include the lack of development tools that provide code validation (of the HTML language), lack of robust error recovery, lack of support for concurrent editing of

---

[6]http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/whats-new.html
[7]http://www.rsa.com/faq/faq_gnrl.html

data, and a lack of modern software development models such as objects or structured programming. To consider the first of these points, realize that development of application systems with HTML will require not only a stable definition of the HTML language, but will demand editors or syntax-checking tools that validate the language constructs when they are written. HTML is a language that is interpreted at run-time by client software, and as such may generate catastrophic errors with limited recovery possibilities. HTML is a platform-independent standard and its success has been determined by the interoperability of servers, browsers, and editors. Each tool must gracefully handle discrepancies in the language stream it processes.

Error recovery is tedious in the most tightly controlled transaction processing systems, especially in those with significant distributed components. An HTML application is highly distributed and communication between nodes may be circuitous. This invites problems in processing both requests and responses. Requiring transactions to be idempotent[8] limits the usability of the medium as an application development tool. In this context, an idempotent transaction is one that can be repeated without impacting the state of the application system. Obviously, a request for a page of hypertext can be repeated without deleterious effects. The same may not be said for a database update in an order-entry system, for example. There are many states that a system may achieve when a request is processed. Of these, successful completion is but one possibility. The area of error recovery in applications systems must be carefully considered, and problems increase as the number of interconnected clients and servers grows.

Web browsers have a one-to-one relationship with a server (for any one request; not necessarily for all links on a page), and the order of requests and responses are well defined. However, to allow applications that can take advantage of application architectures such as concurrent editing (groupware) may well require protocol or design changes to allow either transaction partner to invoke a request. For example, consider a rudimentary shared spreadsheet. If one browser updates a field in a database held by the server, how will a second browser see this information *before* it sends a second update?

---

[8]This is a term for operations that always produce the same result, given the same input. Its use here indicates an application operation that can be performed repetetively, always leaving the system in the same state.

Lastly, development of complicated applications can be taxing with the best of tools. Development of applications with HTML is currently a heavily manual process, from editing HTML pages to creation of scripts which interface with operating system resources. The existence of development tools and development of standard code libraries will make the process more straightforward. These tools and libraries can assist the programmer in manipulation of relevant environment variables (used to send information from the server to operating system scripts) and in creation of more robust pages of HTML output generated by such scripts. Likewise, editors and syntax checkers can help validate the contents of static (not dynamically generated) pages.

This is not an exhaustive list of problems with application development with HTML, but is indicative of the types of problems faced by developers. As an example to elaborate on just one of these problems (language stability), consider the fact that there is currently a wide range of features in the capabilities of various Web clients. Some fully implement the HTML language directives that comprise the forms model used to create interactive applications, while others merely allow passive processing of existing information. Given the development effort underway at this writing, each family of clients should include basic interactive features, but a successful HTML application must target a reasonable minimum feature set. What types of features might a developer employ with confidence that his target audience would be able to utilize all of the application's functionality? Will an application designed for one browser work equally as well (or even work at all) on another? An application developer is writing for a medium in which he controls only a part of the chain between the resource and the user, and extensive testing as well as standard interoperability of language elements is very important.

We will now turn to a brief history and technical overview of the Web to explain the problems the Web was intended to solve and how the current software functions. The Web has grown far beyond the original expectations of the developers and it is being used for many diverse applications. The problems mentioned in the section above must be resolved as the Web continues to grow. However, before we

elaborate on these issues, it is necessary to build upon the reader's understanding of the current model and its strengths and weaknesses.

Chapter 3 - History and Concepts of the World Wide Web

3.1 - History

Developed at CERN and now popularized by work at NCSA, the World Wide Web (WWW) is a client/server TCP/IP-based service providing integrated text, graphics and multimedia. Not only does it allow presentation of static information, but hypertext pages may be created dynamically by the server, either in response to certain stimuli (such as periodic creation of server statistics as an HTML page) or in response to client input. This highly flexible information processing tool may provide the nucleus for many future Internet services.

This project began in March of 1989 at CERN in Switzerland. It was envisioned as a tool to help researchers navigate the various information resources in use by researchers at the European high energy physics laboratory. It was planned as a hypertext tool to facilitate expedient traversal of text, graphics, or data bases. It did not pick up speed, however, for over a year, until the fall of 1990[9]. The first line-mode ASCII browser was demonstrated in late December of 1990, and browsers were first released in limited distribution in the early spring of 1991[10].

The programs from the project were released to the Internet in August of 1991. Development and interest built rapidly. An interface was built for WAIS, and the line-mode browser was available as an anonymous telnet service. The year 1992 saw improvements to the line-mode browsers, but it was not until January 1993 that graphical browsers entered widespread distribution. At this point, the Web contained about fifty HTTP servers and HTTP traffic accounted for about 0.1% of NSFNet[11] backbone traffic.

By September 1993, NCSA (National Center for Supercomputing Applications) had released its *Mosaic* graphical browser for common windowing platforms (Macintosh, Microsoft® Windows™, and X/Windows). The Web numbered approximately 500 registered servers and constituted 1.0% of the NSF

---

[9]http://info.cern.ch/hypertext/WWW/Proposal.html
[10]http://info.cern.ch/hypertext/WWW/History.html
[11]The NSF Net is the "backbone" of the Internet in the United States. It was created with and is supported by the National Science Foundation.

backbone traffic. NCSA released its own server, and the Web continued to grow rapidly. Commercial interest began to increase, and many start-up companies began marketing services or browsers. These commercial browsers may be collectively referred to as value-added browsers, since many browsers are freely available, having been written and placed into free distribution by their authors. There must, therefore, be some additional features or performance/support benefits to induce users to pay. Commercial or not, traffic still continued to grow rapidly, and in June of 1994, there were 1500 registered servers, and probably many more were publicly available but were unregistered. According to a well know *web walker*[12], there were 4,784 servers located in scans of the Internet by August 11, 1994[13]. Tallies from other web walkers topped 7,000 servers by the end of September, 1994. Today, many servers offer multiple home pages[14], often for departments within that organization. Increasingly, there are also servers offering fee-based Internet presence to any group willing to pay for a series of pages. A home page index of the Web had cataloged over 19,000 home pages in December of 1994[15].

---

[12]A web walker or robot is a program that "circumnavigates" the HTTP servers on the Internet, retrieving pages from each as they are located. The archives of URL's (and sometimes full text) are typically indexed for subsequent retrieval with engines such as WAIS. There is considerable research in this area, as well as controversy about the way such searches should be done. Should the robot traverse the web "breadth first", touching many servers at their highest levels, later coming back to load further pages, or should it take a "depth first" search, retrieving many pages from each server it contacts?

[13]http://lycos.cs.cmu.edu/

[14]A *home page* is the top, or entry level for hierarchial access to an organization's WWW resources. In many cases, users will bypass the home page in favor of direct links to relevant resources, but one can in fact locate many organizations on the Web at the time of this writing by simply forming a logical server name for the organization and seeing if a server responds. Convention dictates that home URL's often take the form "http://www.*organization.domain*/".

[15]http://harvest.cs.colorado.edu/harvest

3.2 - Basic Concepts

The World Wide Web is a network service, and it is built on top of existing local and wide area networks. The transport layer of the Web is the protocol HTTP[16]. This is built on top of the reliable transport service provided by TCP, which in turn uses the internet datagram facility IP. The standard implementation of an HTTP server, or daemon, is at TCP port 80. However, it is quite possible for a server to run at another port, and in fact test or development servers will often run at a secondary port address. Additionally, there are certain security implications of running a service at a TCP port numbered below 1024, and this flexibility (of allowing alternate port addressing) permits users without full access to the server hardware to establish web servers.[17]

The basic model of the World Wide Web is straightforward. A client program, using a URL to identify a resource at the server, requests information which is returned in the form of a page of HTML commands. The client program parses this stream of commands and uses tools from its native operating environment[18] to build display images representing the material in the command stream. This command stream from the server may include text formatting information, embedded graphics and other selectable links. The command stream is ASCII text. It is actually possible to interact with a Web server by opening a telnet session to its TCP/IP port and typing a simple command, followed by two carriage returns. For example, the reader may connect to a Web server by using telnet and issuing the command in figure 3.2.1 (followed by two "newline" keystrokes).

Figure 3.2.1:

GET /adirectory/apage.html HTTP/1.0

---

[16]http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html

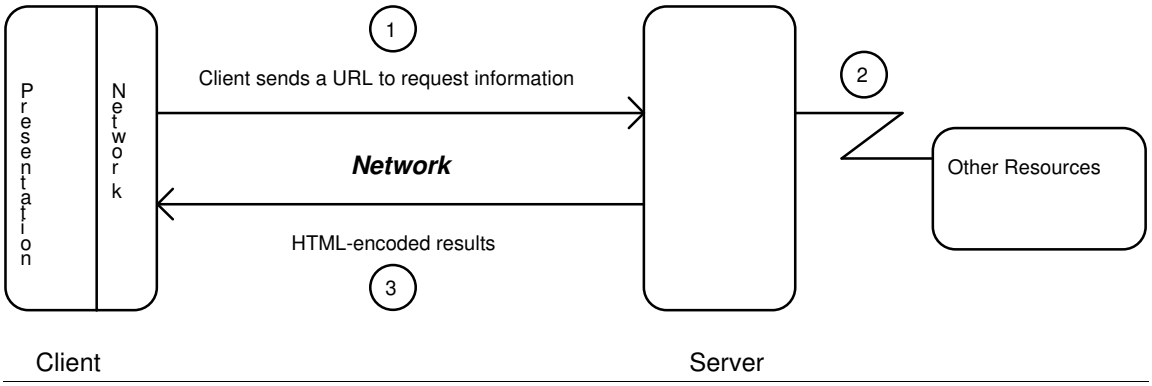
[17]Under UNIX, to run a communications daemon at a port number less than 1024 requires access to the supervisor or "root" account.

[18]The operating environment may be a simple ASCII display or may be a windowing system, such as X/Motif, Microsoft Windows, or the Apple Macintosh.

This will instruct the server to which you have connected to send you "apage.html" from "adirectory", relative to the configured document directory for that server. After the server sends the data to the client, the stateless connection between the client and the server is terminated.

A very simple request and response is shown in the following figure. Note that the communication is initiated by the client, and that the server may optionally access other resources before returning the results.

Figure 3.2.2:



In step one of figure 3.2.2, the client program sends a request to the server. The content of this message is the pathname of the HTML document and a few protocol directives (as in figure 3.2.1). If this is only a request for a page of hypertext, the server will then simply read the file and copy the contents of the file out on the network connection, as indicated in step three of figure 3.2.2. This data stream could look much like the page shown in figure 2.1. However, if the request transmitted in step one names a CGI script, the action of the server is very different. In this instance, the server will perform some action in a script file (step two of figure 3.2.2) before returning the requested information to the client. Here is where the server might access information from a database before returning the HTML-encoded results in step three.

The model of the Web is truly one of heterogeneous client/server networking. Although the initial server was written for UNIX, servers have been adapted for most other major computing platforms[19], including server implementations for personal desktop machines such as the Macintosh and PC's running Microsoft Windows. There are two well-known server daemons for UNIX hosts; the original from CERN and the more recent effort from NCSA. There are other servers for UNIX, but their popularity is limited. This software typically is freely distributed in both source and binary format which encourages experimentation, adaptation, and creates a large community of users who understand many nuances of the software's functionality. There is a quite functional experimental server written in PERL that takes only two-and-a-half pages of source code.[20] This server, *tinyhttpd*, was written by its author, Olaf Titz, to facilitate testing new Web pages. It is also a quite informative overview of server functions and a good learning tool, even including support (although not complete) for CGI scripts. *Tinyhttpd* as distributed (version 1.4, 1994/08/15) is not a concurrent server, but during my research I have added that functionality with only a handful of additional lines of code to create a concurrent version of this program. In general, all these web servers and the client browsers are fully interoperable, although there are a few features that are not supported in all implementations.

Installation and activation of a server is straightforward assuming the proposed host already has a working TCP/IP connection to the Internet. As mentioned above, server distributions for many computing platforms are freely available on the Internet. The first step is to copy the code from an FTP site, taking care to obtain a recent copy of the server you are planning to use. It is probably easier to begin with a pre-compiled binary if one can be located for your platform. I use the NCSA UNIX daemon, and it is installed by expanding its archive file[21] to create the basic subdirectories for configuration files and template CGI scripts. To activate the server, it is necessary to make changes in the configuration files to reflect the directory structure of the host, the user ID which will control the daemon and the TCP port (if the user elects to use a non-standard port). Configuration is a simple task and can be accomplished in a few minutes.

---

[19]http://info.cern.ch/hypertext/WWW/Daemon/Overview.html
[20]http://rzstud1.rz.uni-karlsruhe.de/~uknf/sw/tinyhttpd.html
[21]The archive files are typically distributed in TAR format for UNIX computers.

In the sample configuration file (from the NCSA server) included below in figure 3.2.3, note that the *User*, *Group*, *ServerAdmin*, and *ServerRoot* parameters have been changed for the installation.

Figure 3.2.3

```
# This is the main server configuration file. It is best to
# leave the directives in this file in the order they are in, or
# things may not go the way you'd like. See URL http://hoohoo.ncsa.uiuc.edu/
# for instructions.

# Do NOT simply read the instructions in here without understanding
# what they do, if you are unsure consult the online docs. You have been
# warned.

# Rob McCool (comments, questions to httpd@ncsa.uiuc.edu)

# ServerType is either inetd, or standalone.

ServerType standalone

# If you are running from inetd, go to "ServerAdmin".

# Port: The port the standalone listens to. For ports < 1023, you will
# need httpd to be run as root initially.

Port 80

# If you wish httpd to run as a different user or group, you must run
# httpd as root initially and it will switch.

# User/Group: The name (or #number) of the user/group to run httpd as.

User www
Group infoserv

# ServerAdmin: Your address, where problems with the server should be
# e-mailed.

ServerAdmin longlegs@bullhead.adp.unc.edu

# ServerRoot: The directory the server's config, error, and log files
# are kept in

ServerRoot /home/www

# ErrorLog: The location of the error log file. If this does not start
# with /, ServerRoot is prepended to it.

ErrorLog logs/error_log

# TransferLog: The location of the transfer log file. If this does not
# start with /, ServerRoot is prepended to it.

TransferLog logs/access_log

# PidFile: The file the server should log its pid to
PidFile logs/httpd.pid

# ServerName allows you to set a host name which is sent back to clients for
#your server if it's different than the one the program would get (i.e. use
#"www" instead of the host's real name).

#ServerName new.host.name
```

The following NCSA server configuration file controls the location of the document directory, script directory, and also allows the server administrator to add data types that are recognized by this server. In the following example, the values for *DocumentRoot* and *ScriptAlias* have been changed for the local environment.

Figure 3.2.4

```
# With this document, you define the name space that users see of your http
# server.

# Rob (robm@ncsa.uiuc.edu)


# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.

DocumentRoot /home/www/htdocs

# UserDir: The name of the directory which is appended onto a user's home
# directory if a ~user request is recieved.

UserDir public_html

# DirectoryIndex: Name of the file to use as a pre-written HTML
# directory index

DirectoryIndex index.html

# AccessFileName: The name of the file to look for in each directory
# for access control information.

AccessFileName .htaccess

# DefaultType is the default MIME type for documents which the server
# cannot find the type of from filename extensions.

DefaultType text/plain

# AddType allows you to tweak mime.types without actually editing it, or to
# make certain files to be certain types.
# Format: AddType type/subtype ext1

AddType application/winword doc
AddType application/excel xls xlc slk
AddType application/winhelp hlp
AddType application/exe exe

# Redirect allows you to tell clients about documents which used to exist in
# your server's namespace, but do not anymore. This allows you to tell the
# clients where to look for the relocated document.
# Format: Redirect fakename url


# Aliases: Add here as many aliases as you need, up to 20. The format is
# Alias fakename realname


# ScriptAlias: This controls which directories contain server scripts.
# Format: ScriptAlias fakename realname

ScriptAlias /cgi-bin/ /home/www/cgi-bin/
```

There are other configuration files in the NCSA server implementation, such as an access restriction file and a data type configuration file, but the server operates well with the default values in these files until the server administrator becomes proficient and needs to change or add functionality. After configuration, the server is actually loaded by execution of the daemon. If there are HTML pages in the server's document directory[22], they can then be retrieved as in figure 3.2.1. The NCSA server as distributed includes several CGI scripts that can be used to validate an installation.

Client software for the Web is even more broadly available than server code, with several different browsers often available for each popular platform. The basic UNIX browser is the original line-mode browser from CERN. Another character-based browser for UNIX is *Lynx*, but significantly *Lynx* adds support for interactive forms[23], which are essential for application development. The most well know UNIX browser, however, is the graphical client *Mosaic*. In fact, in many cases, the name *Mosaic* has almost become synonymous with the Web much as the term *xerox* has come to mean photocopy. There are versions of *Mosaic* available for Microsoft Windows and the Macintosh as well, further adding to the popularity of this family of browsers. Other Macintosh browsers include *Netscape, MacWeb* and *Samba*. For Microsoft Windows, the *Cello* browser from Cornell Law School is common[24], as is the browser *WinWeb* from Enterprise Integration Technologies (EIT). On all platforms, however, the preferred browser at the time of this writing may well be *Netscape*[25] from Netscape Communications Corporation. Its initial beta release in October 1994 was for virtually all major client environments, and it provides significant improvements in performance, stability, and usability. The November 1994 *Netscape* beta release continued to build on these successes, fixing bugs and continuing to add functionality. Significantly, this

---

[22] The document directory is defaults to the directory *htdocs* relative to the *ServerRoot* directory. This is set in the "srm.conf" configuration file in figure 3.2.4.
[23]Interactive forms are the principal building blocks for HTML application development. Forms are comprised of a set of HTML tags that enable the user to enter data into a stream to be sent to the server for processing. Forms are discussed in detail in Chapter 5.
[24]Version 1 of *Cello* does not support forms, however, limiting its usefulness in application development. The developer of *Cello* has version 2, including forms support, in alpha testing at this writing.
[25]http://home.mcom.com/MCOM/products_docs/client.html

release (beta 0.93) offered secure communications to Netscape Communication Corporation's own modified server, *Netsite*[26]. *Netscape's* official 1.0 version was released in late December, 1994.

One emerging trend in client development is the privatization of Web browsers. Over the first years of the Web's existence, virtually all browser programs have been developed at research or educational institutions and have been freely distributed. This has, in fact, significantly helped to make the Web what it is today. Practically anyone with a network connection could have access to these tools. However, as the expectations for the Web have grown, so has the realization that loosely supported tools would not easily make the transition from academia into the corporate world. In 1994 we have seen the first commercial vendor-supported browsers, such as the *Air Mosaic* package from Spry, Inc. and *NetCruiser* from Netcom Systems. Quadralay Corporation has a browser called *GWHIS* (Global-Wide Help & Information System). *WinWeb*, mentioned above, is a "quasi-commercial" browser, available in both supported and unsupported forms. Also, the browser *Netscape* from Netscape Communications Corporation fits this latter model. It is available for royalty-free use by the academic community, but commercial use requires a license payment. Additionally, Netscape Communications Corporation, like most such vendors, offers support and consulting services that may be contracted separately. These commercial browsers provide enhanced end user support, and also attempt to add value through features enhancing usability. As an example, *NetCruiser* has an interface with a graphical "Internet Site Chooser", which is essentially a map configured with commonly accessed resources. *Netscape* offers dramatic speed enhancements on slow network links (such as analog phone lines) as well as an improved user interface. As mentioned above, this generation of browsers is also beginning to offer enhanced security features, though they are still often proprietary. In fact, the Spry browser, *Air Mosaic*, is available as a demonstration product which an interested customer can purchase on-line by entering his credit card number. This is a secure transaction because Spry has modified their browser and their own web server to encrypt the transmission. This is not yet available in the general case, as we will explore later, in section 9.3.

---

[26]http://home.mcom.com/MCOM/products_docs/server.html

For additional background information on the topics discussed above as well as current developments on the Web, the interested reader may wish to consult the World Wide Web's Frequently Asked Question (FAQ) file.[27] The material in the FAQ contains a collection of references (in hypertext form) to many of the links quoted in this paper, and is a good starting point for Web novices as well as a resource for the experienced user. The FAQ covers the background of the Web, servers and browsers, CGI programming, security, and many other relevant topics.

[27]http://sunsite.unc.edu/boutell/faq/www_faq.html

Chapter 4 - World Wide Web Browsers

Now that the reader has a basic exposure to the Web and its concepts, we will focus on a further explanation of the Web clients, or browsers. First, the reader should understand that in addition to interpretation of HTML, the Web browser is able to process other information resources. Perhaps the most widespread type of information server is still the Gopher[28] server, and many types of information are still available only in that format. The web browser typically processes Gopher menus like a native Gopher client. The level of support for recent Gopher features may not be as complete as a native client, but most information is retrievable, and the full-featured Web client will usually be able to launch external viewers with retrieved Gopher documents. The Gopher information model relies on hierarchical presentation of menus, rather than hypertext links. The importance of Gopher support may be rapidly diminishing, however. The growth of the Web has been so explosive that many institutions that have employed Gopher servers for information dissemination are quickly implementing Web servers and converting the data from Gopher menus to Web pages. Gopher support may soon be used only to access legacy systems.

Another common information resource on the Internet is the WAIS server. Web pages and Gopher menus present information in a form that the user of a Web browser can navigate, either in a linear or hypertext mode. WAIS databases are *indexed* collections of information. Where Gopher presents information in a linear fashion and Web pages provide hypertext links, WAIS provides indexing services and document retrieval. This may range from indices of text to indices of other files and data objects. In fact, many large Web servers index their HTML pages with WAIS to allow their users to quickly locate entry points relevant to particular search paths. The different objectives of WAIS and the Web encourage interoperability of these complementary tools. The Web needs the functionality of WAIS but not of Gopher, which the Web is absorbing. It is this integration of HTML and support for links to other information servers that creates much of the appeal of the World Wide Web.

---

[28]gopher://gopher.tc.umn.edu:70/1

In the World Wide Web, browsers are usually larger and more complex than the servers. Recall that in Section 3.2 we discussed a minimal server written with just over two pages of instructions in the PERL language. Conceptually, the server's basic job is simple: it provides a stream of data in response to a request for that specific data instance (a URL). The browser must render the information to the display environment where it executes; the more complicated the display environment, the more ambitious the task. With WAIS, the protocols that pass between the client and server are complex. There is a close association between the code base in the client and server environments, and consequently full WAIS support has been slower to appear in Web browsers than Gopher support. Of the *Mosaic* family of browsers, for example, only the X version supports native WAIS queries, due to the availability of public domain code libraries for UNIX that can be employed to provide WAIS capabilities. Other Web browsers (including the Macintosh and Windows versions of *Mosaic* and *Netscape*) must use gateways[29].

One important part of a browser's job is to launch external viewers. This can add multimedia functionality to applications developed for Web browsers. A server daemon is configured to tell its clients what type of data it is sending. Some directives may instruct a client to render an image from a graphic file in Graphics Interchange Format (GIF) and embed this image directly in the page, but other links from a page may direct a client to copy similar information via HTTP and launch an external viewer. For example, a server may send a graphic image (in a format such as GIF or the format defined by the Joint Photographic Experts Group (JPEG)). The browser can then either save the file or dynamically start a viewer program that can manipulate information in this format. This is often seen through the practice of using reduced views of pictures as a menu and sending the full image only if explicitly selected. Other common types are audio and video. Generally speaking, the server and client cooperate through use of Multimedia Internet Mail Exchange (MIME) formats, which define data types and allow browsers to be pre-configured for

---

[29]A gateway between the Web and information resources such as WAIS servers usually takes the form of a system written as CGI scripts. These scripts are accessible directly from the Web, and they contain logic to communicate with other servers that do not speak HTTP. An example of this will be discussed in Chapter 8.

certain common viewers. These MIME types are defined by standards, but most servers and browsers allow this model to be extended. This has interesting ramifications for the developer of HMTL applications. For example, an HTML page can be used as a menu to launch applications on the local system. A spreadsheet could be distributed as a part of an Executive Information System by sending it from a server to the browser, and letting the browser launch the spreadsheet's native application. Word processing documents, help texts, or business-specific applications can be invoked. A server and browser may even be configured to copy an executable program from the server and launch it from the browser, provided that it is in the correct machine language for the operating environment at the browser.

Whatever type of information it processes, the Web browser tries to make server-to-server transition seamless. Whether all reference materials are on the same or different Web servers, on Gopher servers, on WAIS servers or on local client systems, the Web browser attempts to make all data appear to be at the fingertips of the user, who in fact is often not aware that he has left one server and is perusing another. Simply put, the object of the Web browser is to provide a single tool that can be used to find and utilize all information resources that appear on the Internet.

Chapter 5 - What is an HTML Application?

As mentioned in Chapter 3, the language of the World Wide Web, HTML, is a tag-oriented markup language derived from SGML[30]. It is composed of pairs of tags, where the start tag is always required, but the end tag may often be omitted. It is much like any markup language that employs coded directives to direct generation of formatted output. For example, there are tags for the title of a page to be displayed in a browser's title bar or other conspicuous location. This is coded with a syntax such as "<TITLE> *put a title here* </TITLE>". Likewise, there are codes for text highlights, heading levels, lists of items, and other text emphasis or stylistic attributes. The actual display format of these codes is often determined by parameters provided to the browser. This material will not be covered in detail here, and the interested reader should refer to the NCSA documentation on this subject for an effective tutorial[31]. Some features do require more explanation since they provide the fundamental tools for application development, which is the focus of this discussion. They will be discussed in more detail after the basic definition for the framework for application development is created. However, here is a brief explanation of some common directives that appear in many pages and that are found in example 5.1:

Figure 5.1:

<TITLE>...</TITLE>	Defines the title of the page, used in broswers to describe the page in bookmarks or history lists, for example.
<H1>...</H1>	A first level heading; this will use a large font and usually bold face.
<A HREF=*specify URL here*>...</A>	A hypertext link to another Web page or multimedia file (such as a picture, sound, or movie). If the link is to a multimedia file, the browser will often launch an external viewer to process the contents of the file in a cooperating program. It is possible to have text or an image as a link; in the example below you will see the *<IMG SRC>* directive as the link that appears on the page. This means that the picture itself describes a link to another resource. This feature is used to help create context sensitive image maps, and this will be discussed later in this chapter.
<P>...</P>	A paragraph break. The closing delimiter is usually omitted.
<IMG SRC=*specify URL for picture here*>	This describes a picture to be embedded directly in the page as it is shown in the browser. Such an image differs from a image specified as a hypertext link (anchor) described with the <A>..</A> directive.
<HR>	A "horizontal rule". This creates a line across the browser's display to separate logical parts of a document.
<PRE>...</PRE>	Preformatted text, where the information between tags retains its spacing and line breaks. This is useful in displaying output from system commands, for example. However, HTML directives embedded in this stream are still processed, allowing pictures to be inserted, for example.

[30] http://info.cern.ch/hypertext/WWW/MarkUp/Intro.html
[31] http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html

To understand application development with HTML, it is first important to understand non-interactive pages. As an example of a typical Web page, first consider the author's *home page*[32]. Standard practice is to make such a home page a multimedia extension of the UNIX *.plan* file, complete with information about the author and local or remote references to items of professional or personal interest to the author.

Example 5.1:

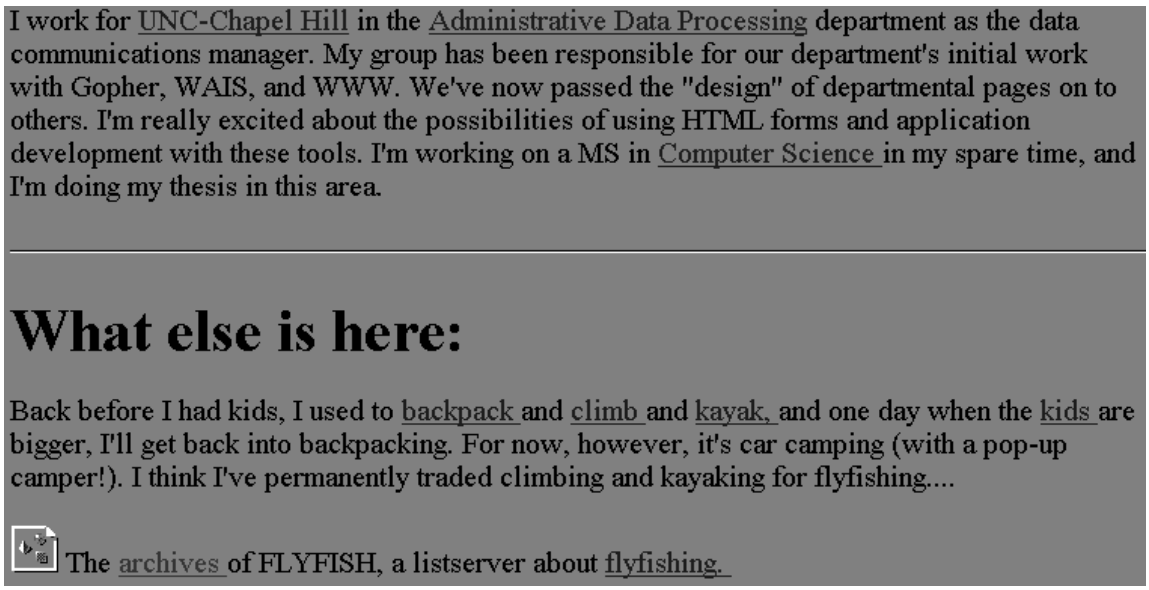
```
<Title>Joel's Home Page</TITLE>

<H1>Welcome to Joel Dunn's Home Page</h1>

<a href=http://www.adp.unc.edu/~longlegs/boonfork.jpg> <img src=http://www.adp.unc.edu/~longlegs/bfork16.gif></a>

<P>Here is my page....I like flyfishing, so this I've included this picture of me in action on Boone's Fork in the North Carolina mountains. Sorry to say I didn't catch any <a href=http://www.adp.unc.edu/~longlegs/fish.html>fish </a>on this trip, but it was a beautiful day. I did miss a couple of <a href=http://www.adp.unc.edu/~longlegs/fish.html>fish </a>that hit a Royal Stimulator. My wife took this picture on a <a href=http://www.adp.unc.edu/~longlegs/family.html>family </a> hike. The original snapshot is in color, and if you select the icon above, you'll get the color picture as a 987x795x256 JPEG file of 290842 bytes...This is a new copy of this file as of 9/23/94, with improved shading...the scanned copy was a little dark so I lightened it up a bit and I think it looks more like the original.
<P>
<hr>
I have re-scanned the pictures on the second-level pages. They are bigger (byte-wise) but the results are better. Also, I've uploaded jpeg versions that you get when you click an individual picture. Enjoy! I hope to get some more pictures, especially fishing pictures, soon. I've got a couple of fishing trips coming up!
<p>
<hr> I work for <a href=http://www.unc.edu/> UNC-Chapel Hill</a> in the <a href=http://www.adp.unc.edu/> Administrative Data Processing</a> department as the data communications manager. My group has been responsible for our department's initial work with Gopher, WAIS, and WWW. We've now passed the "design" of departmental pages on to others. I'm really excited about the possibilities of using HTML forms and application development with these tools. I'm working on a MS in <a href=http://www.cs.unc.edu/>Computer Science </a>in my spare time, and I'm doing my thesis in this area.<P>
<hr>
<h1>What else is here:</h1>

Back before I had kids, I used to <a href=http://www.adp.unc.edu/~longlegs/backpack.html>backpack </a>and <a href=http://www.adp.unc.edu/~longlegs/climb.html>climb </a>and <a href=http://www.adp.unc.edu/~longlegs/kayak.html>kayak, </a>and one day when the <a href=http://www.adp.unc.edu/~longlegs/family.html> kids </a> are bigger, I'll get back into backpacking. For now, however, it's car camping (with a pop-up camper!). I think I've permanently traded climbing and kayaking for flyfishing....
<p>
<img src=http://www.adp.unc.edu/~longlegs/redball.gif>
The <a href=http://www.adp.unc.edu/cgi-bin/wais-flyfish-q> archives </a> of FLYFISH, a listserver about
<a href=http://www.geo.mtu.edu/~jsuchosk/fish/fishpage> flyfishing. </a>
<p>
<img src=http://www.adp.unc.edu/~longlegs/redball.gif>
The <a href=http://io.datasys.swri.edu/> Backcountry </a> page
<p>
<img src=http://www.adp.unc.edu/~longlegs/redball.gif>
<a href=http://www.adp.unc.edu/cgi-bin/finger> Finger </a>somebody!
<p>
<img src=http://www.adp.unc.edu/~longlegs/redball.gif>
<a href=http://www.adp.unc.edu/cgi-bin/finger?longlegs> Finger me!</a>
```

[32]The author's home page is available on the Web as "http://www.adp.unc.edu/~longlegs/". It is difficult to render this page effectively into a printed document such as this one. The reader interested in the way this page looks in a browser should access the URL with a browser compatible with his workstation. Web pages are changeable, however, and this page like most others is in a continual state of flux. There may be substantial differences between this page as described in this paper and its appearance when the reader views it. In this case, examine the "source" (a display of the raw HTML that is a function in many browsers) and compare that to the page's representation.

```
<p>
<img src=http://www.adp.unc.edu/~longlegs/redball.gif>
<a href=http://www.adp.unc.edu/cgi-bin/whof> Who's </a> home (with finger)?
<p>
<img src=http://www.adp.unc.edu/~longlegs/redball.gif>
<a href=http://www.adp.unc.edu/cgi-bin/wais-html-q> What's </a> on this Web server?
<p>
<hr>
If you want my PGP public key, finger me using the option above...

<pre>

This page is maintained by <a href=http://www.adp.unc.edu/~longlegs/mymailform.html>joel_dunn@unc.edu</a>

Last updated 23-September-1994
</pre>
```

A partial rendering of this page is displayed below. This has been cropped to display a portion of the screen, starting with third paragraph (third <*P*> tag) in example 5.1 above. A comparison of this with the directives in example 5.1 should give the reader a flavor of the way such a page might be seen with a graphical browser.

Figure 5.2:



What is important to see here is that the page is composed of text interspersed with many varied language elements. It may present only text or may include multimedia objects, such as images, audio clips, and movies. Remember that the Web is a hypertext system, and pages may indeed be created and modified dynamically. Linkages between systems or to other pages on the same server are provided by the directive pair "<A> *hypertext link* </A>". Above, note that this can be seen in the reference to the flyfishing page, for example. This is a reference to a page on a remote server. Also, note that in this display, hypertext

links are shown with underlines. The browser might also use different colors to further enhance the distinction between regular text and hypertext. The hypertext links can be to other local information pages, or they can direct the Web browser to a remote application system or set of pages.

A major feature that differentiates HTML from static markup languages is the "<FORM> ... </FORM>" identifier. This signifies that the language elements contained within the forms identifier will define a screen that takes user input in response to action elements encoded on the form. Input objects on a form may include data entry fields and many graphical features such as "drop-down" list boxes, scroll bars, check boxes, and radio buttons on Web clients operating in a graphical environment.

These elements alone might provide nothing but the facility to change the appearance of forms and traverse various informational links if it were not for the ability of Web servers to process data collected from a browser and invoke an operating system script file with the parameters gleaned from the form. This is called the Common Gateway Interface (CGI) in the HTTP daemon. This combination -- interactive data input facilities and operating system command interfaces -- come together to provide the tools necessary to build application systems.

To begin our exposition of application development, let's look at an artificially simple example. This HTML page prompts the user to enter a value into a field. A browser would initiate this conversation by requesting this form from the server, naming the page with a URL. The browser will display the text with a data entry area. When the user has keyed a value, he may send it to the server where its value will be echoed back to the client. The line numbers on the right of the example are for instructional use and would not appear in the actual form.

Example 5.2:

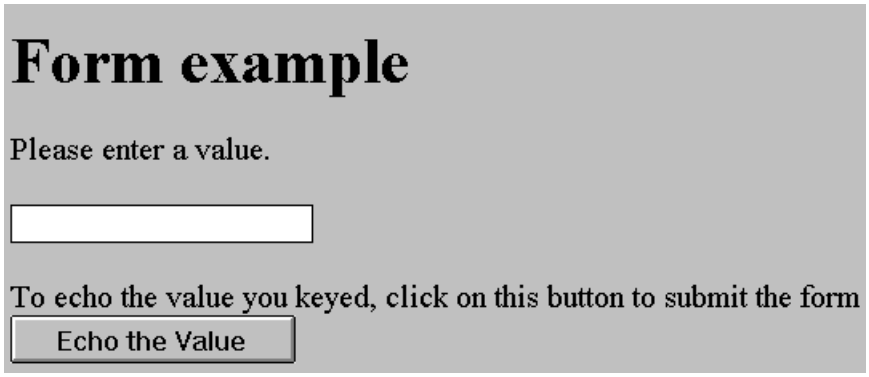
```
<TITLE>Example of basic form</TITLE>                                    (1)
<H1>Form example</H1>                                                   (2)
<FORM METHOD="POST" ACTION="http://www.adp.unc.edu/cgi-bin/echo.sh">    (3)
Please enter a value.<P>                                                (4)
<INPUT NAME="Entry"><P>                                                 (5)
To echo the value you keyed, click on this button to submit the form    (6)
<INPUT TYPE=submit VALUE="Echo the Value"><P>                           (7)
</FORM>                                                                 (8)
```

If we examine the page one line at a time, we will see which parts are cosmetic and which parts are actually a part of the application. First, line 1 is the title of the form. As we have seen before, this is used to identify this page in a browser. Line 2 is a heading, used to delineate a major section of the page. This has also been discussed earlier. However, on line 3, we introduce a new construct. We have alluded to the importance of the <*FORM*> tag, and here we see it in use. This tag has many options, but this usage in line 3 is by far the most common that the reader will encounter in perusing actual applications. In fact, most application systems use coding like line 3 changing only the *action* value as appropriate within the context of the system. Simply put, this line instructs the browser to request the URL specified with the *action* keyword and to package up any data entered on the form in a format called "URL-encoding". URL-encoding is basically a datastream with the field names and their values, where spaces and certain non-printable fields are converted to a hexadecimal representation[33]. Line 4 is once again simple text that provides the user with instructions to fill out the form. Line 5, however, is also a relevant part of the application. This is the actual field definition and prompts the browser to create an input field internally named "entry". An input field by default is 20 characters wide, but it contains as much alphanumeric text as the user wishes to enter, showing the information in a sliding 20-character window. Line 6 is once again descriptive text. Line 7 instructs the browser to create a button to allow the form to be sent to the server. The text in the *value* keyword appears on the body of the button as it is presented on a graphical browser. Finally, line 8 defines the end of the form. Figure 5.3 below shows a rendering of the image produced by the HTML in example 5.2.

---

[33]For example, the string "foo bar" would be URL-encoded as "foo%20bar". The character % is used as an escape sequence to introduce the hexadecimal representation of the character. This encoding is used to eliminate problems such as passing parameters with embedded spaces, for example.

Figure 5.3



If we imagine that you, the reader, have now accessed the form in example 5.2, you would have the form above displayed in your browser, prompting you to key a value. However, this form alone cannot do anything with the value you enter. We need to look at the script specified as the *action* URL. When you enter your data into the input field, that data is held locally in the browser until you are ready to send it to the server. When you submit the form, the field name and value are appended to the *action* URL, and are made available to the CGI script on the server. As in the previous example, the line numbers on the right in this script are for explanatory purposes only and are not a part of the script.

Example 5.3:

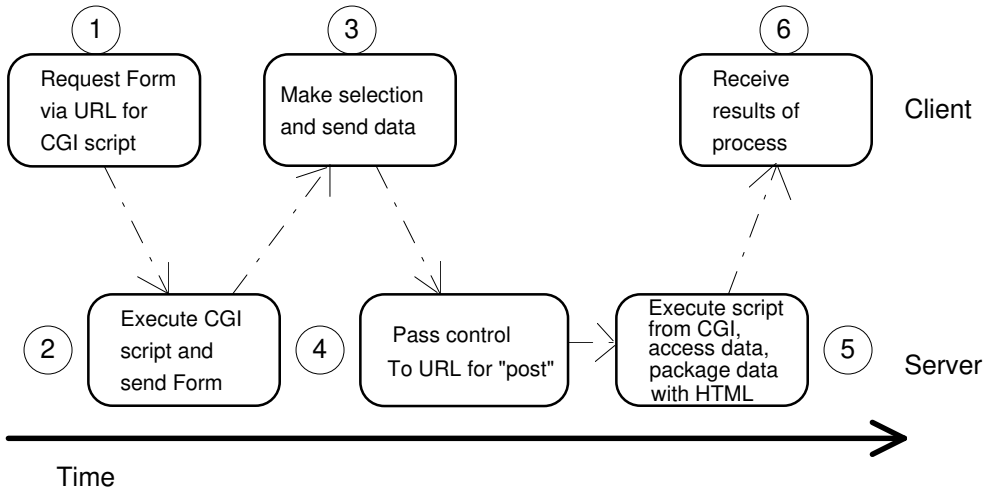
```
#!/bin/sh                                                    (1)
echo Content-type: text/html                                 (2)
eval `/home/www/cgi-bin/cgiparse.run $CONTENT_LENGTH`                (3)
echo \<H1\>Echo Output\</H1\>                                (4)
echo $Entry                                                  (5)
```

Example 5.3 shows a UNIX shell script which must first identify the shell that is to process the script. This is the function of line 1, and this is a trait shared with all UNIX shell scripts. Line 2 is a Web-specific function. When this script is run, its output goes to the standard output device which is actually a socket connection to the browser. Therefore, the first output from the script must prove to the browser that the output stream is valid. This is the function of line 2. The statement at line 3 invokes a program that extracts the field value from the browser's input stream and creates an environment variable for later use in the script. Such parsing is a standard function on the Web and there are many programs that can perform

this task. Further discussion and a reference for additional background on field parsing is included later in this chapter. Continuing with the example, line 4 sends an HTML tag to create a heading. Note that the *title* tag is not present, since it is not mandatory. Line 5 actually returns the input value to the browser. The name *$Entry* for the variable in this line is a direct result of the field name (*Input Name="Entry")* also on line 5 in the example 5.2. The parsing program in line 3 uses that value from the URL-encoded stream to create the variable name.

The previous example was created solely to explain the basic steps in an HTML application. Let's take a look at a slightly more complicated system now. This is a simple but real example that is available on the author's home server. In this example, we will see a data entry form built dynamically by the server with a variable amount of information. The code presented in the examples below is written for the NCSA HTTP daemon using the CGI. It builds a form listing users on the host (at the time when the script is executed) and allows the client to selectively finger a specific user. First let's examine the application flow that will be used in this system. Remember that unlike the first example, the initial HTML page processed by the viewer must be created by the server, since it may be different each time this page is requested.

Figure 5.4:



The script referenced in the original URL (to invoke the system, in step 1 above) is as described in example 5.4. Remember that the function of this script is to create a page of HTML which will prompt the

user for input. In example 5.4, the script construct "cat << EOM" is used to read the characters up until EOM (end of medium) as data and thus avoid direct interpretation of special characters (such as *<, >,* and */*) by the UNIX shell program. Note that where this is not possible, such as in some script lines using *echo*, it is necessary to prefix special characters with an escape sequence.

Example 5.4:

```
#!/bin/sh
#
# Joel Dunn - Last modified October1994

WHO=`which who`

echo Content-type: text/html
echo

if [ -x $WHO ]; then
        cat << EOM
<TITLE>Who's Home? (and finger them)</TITLE>
<H1>Who's Home? (and finger them)</H1>

<FORM METHOD="POST" ACTION="http://www.adp.unc.edu/cgi-bin/finger_sel.sh">

This will show who's home at Bullhead!<P>
<SELECT NAME="User" SIZE=5 >
EOM
                $WHO | cut -f1 -d' ' | awk '{ printf "<OPTION> %s\n",$1 }'
                echo \</SELECT\>
                echo \<P\>
                echo To finger someone, press this button:
                echo \<inPUT TYPE="submit" VALUE="Finger"\>.
                echo \<P\>
                echo \</FORM\>
        else
        echo Cannot find who on this system.
fi
```

This script (executed at step 2) creates HTML which is interpreted by a graphical browser to create a select box with five lines and integrated scroll bars on the right margin. Unlike the first simple example, this page is not static. It changes as the number of users on the host changes, and as such this page must be generated dynamically, requiring this extra step. When this script is executed, it might produce HTML as follows (with only one user, *longlegs*, logged onto this system):

Example 5.5:

```
Content-type: text/html

<TITLE>Who's Home? (and finger them)</TITLE>
<H1>Who's Home? (and finger them)</H1>

<FORM METHOD="POST" ACTION="http://www.adp.unc.edu/cgi-bin/finger_sel.sh">

This will show who's home at Bullhead!<P>
<SELECT NAME="User" SIZE=5 >
```

```
<OPTION> longlegs
</SELECT>
<P>
To finger someone, press this button:
<inPUT TYPE=submit VALUE=Finger>.
<P>
```

The box shown below in figure 5.5 is generated by a graphical browser (*Mosaic)* processing the HTML in example 5.5. It contains the names of the users logged into the host running the HTTP daemon. Additionally, note that the script in example 5.4 has generated the HTML directives to create a button which returns the selected data to the server when it is activated by a mouse click. This corresponds to step 3 in figure *5.4*. On screen, the form looks like this:

Figure 5.5:

When the user selects a target for the finger command (using a mouse click on a listed user ID) and presses the *finger* button, the browser transfers a field containing the form's variable names ("User", in this example) and values (using a URL-encoded datastream) to the URL in *METHOD="POST" ACTION = "http://www.adp.unc.edu/cgi-bin/finger_sel.sh"* (step 4, figure 5.4). This identifies the program or script which parses the incoming data. For the sake of brevity, the full details of the parsing mechanism are again omitted here, and are performed in the program *cgiparse.run* (called in the script in example 5.6 below), which creates environment variables with the values entered at the client. However, to briefly describe the process, *cgiparse.run* is a C program that reads the input stream from the client and then writes the variable name and value extracted from the URL-encoded standard input to standard output. In the Bourne Shell, the syntax "name=value" creates an environment variable, and this is precisely what is emitted by the program *cgiparse.run*. An interested reader may learn more about the parsing process and its many options

from the NCSA server itself[34]. The script *finger_sel.sh*, which is initiated when the data is returned to the server, is shown below:

Example 5.6:

```
#!/bin/sh
#
# Joel Dunn - Last modified October 1994

echo Content-type: text/html
echo

eval `/home/www/cgi-bin/cgiparse.run $CONTENT_LENGTH`

        cat << EOM
<H1>Finger Output</H1>
<PRE>
EOM
finger $User
echo \</PRE\>
```

This issues the operating system "finger" command and embeds the results in the HTML sent to standard output by this script. This entire process results in a formatted display on the browser containing information output by the finger command (step 6, figure 5.4). Interestingly, if a user of a system that supports use of *plan* files chooses to further embed HTML directives in the plan, that information is interpreted by the browser in accordance with HTML syntax and may indeed be a page of HTML itself. Such a *plan* file is still usable outside the context of the Web. If a finger command is issued from a program that does not support HTML (the basic finger tool, for example), the HTML directives contained in the *plan* file simply appear as text in the output of the command. It may be somewhat confusing to one who is not familiar with HTML, but the HTML directives can be placed within the *plan* file so as to permit readibility while still functioning normally when read into a Web browser.

As another example, the data passed to the script in example 5.6 could instead be passed to a database engine such as an SQL server. Properly written data base commands can retrieve information or even add and update records in a data base. There are, however, extensive security considerations for such applications that will be discussed later, and there are potential problems with concurrency as well as

[34]http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html

transaction atomicity. For the time being, however, we will ignore these points and concentrate on how such a system could be constructed.

At this point, realize that the output from an operating system command or data base access routine could be manipulated and written out with embedded HTML directives to produce whatever formatting was desired by the programmer. Recall that in example 5.6, this is precisely what happened with the output from the *finger* command. Now, to expand on this idea of data base access, let the reader consider the following sample application which retrieves information from an SQL server. The flow of this application follows the same model as the finger application, but the script invoked in the parsing routine initiates a data base query. The initial script is as follows:

Example 5.7:

```
#!/bin/sh
#
# Joel Dunn - Last modified October 1994

echo Content-type: text/html
echo

cat << EOM
<TITLE>Who's in the Email Data Base?</TITLE>
<H1>Who's in the Email Data Base?</H1>

<FORM METHOD="POST" ACTION="http://www.adp.unc.edu/cgi-bin/unc_number.sh">

Enter your UNC Number ---> <inPUT NAME="unc_numb">
<P>
<inPUT TYPE="submit" VALUE="Look Up">
</FORM>
EOM
```

Note the structural similarity with the finger application. The following script invokes the parsing routine as seen in examples 5.3 and 5.6 to extract the key for the database retrieval.

Example 5.8:

```
#!/bin/sh
#
# Joel Dunn - Last modified October 1994

echo Content-type: text/html
echo

eval `/home/www/cgi-bin/cgiparse.run $CONTENT_LENGTH`
```

```
        cat << EOM
<H1>SQL Query Output</H1>
<PRE>
EOM
/home/www/cgi-bin/unc_number.sh2 $unc_numb
echo \</PRE\>
```

For the sake of clarity and modularity to isolate the SQL query in a separate script, the code in example 5.8 further calls another script, passing the parsed value entered in the form, and then in turn actually calls the query interface program. Note that the script above in example 5.8 contains HTML directives to provide rudimentary formatting for the query output. The script below in example 5.9 contains the actual query, invoked through the *sqlmain* executable. A script of this nature often has a database user ID and password hard-coded into the text. Below, those constants are shown in italics after the "LOGIN" and "PASSWORD" keywords. The last two lines in this example set environment variables peculiar to a Sybase SQL server interface.

Example 5.9:

```
#!/bin/sh
/home/www/gsql/sqlmain "SELECT amdbemail_t..unc_people.last_name FROM \
amdbemail_t..unc_people WHERE amdbemail_t..unc_people.unc_id = '$1'" \
"LOGIN my_login" \
"PASSWORD my_password" \
"SYBASE_ENV SYBASE=data_path" \
"SYBASE_ENV2 DSQUERY=server_name"
```

The first example (5.2 and 5.3) was intentionally simple and was used to introduce the concepts without excessive detail. We have added features and concepts with the *finger* example and the *SQL* example, but we have still avoided many complications. In fact, the output of one command could build complex display items such as list boxes or drop-down boxes populated with information gleaned from the data base, and the HTML directives could request that the browser pass the information from this dynamic form to yet another parsing engine and access routine. The chain of references started with one such entry point (a URL) could lead the user's browser client to many different logical or geographically dispersed host computers. The programmers who create such applications could move resources manually from one site to another or even automate the process by dynamically building the URL's that provide the action for the *POST* method of forms processing.

From these examples the reader can see that it is quite possible to develop interactive applications. Application development is done with HTML directives embedded in ASCII text, supported by scripts that execute on the server. It is a function of the client to interpret these HTML instructions using the look and feel of its own native environment, but it is certainly necessary to have server-side script processing (CGI) to create a meaningful application. Now, let's turn to further detailed discussion of the features introduced in chapter 5 and other additional features that support application development.

Chapter 6 - Application Development

6.1 - Constructs in Application Development

To support application development, the architects of HTML have included many types of on-screen display tools that allow sophisticated user interfaces to be created with a minimum of programmer effort. What follows here is a synopsis of some of the more important language elements that are commonly found in HTML applications. These features can be examined interactively through the forms tutorial at NSCA[35]. HTML is not case sensitive, and in the following examples, upper and lower case is often used to emphasize certain constructs.

First, consider the standard option box. This was seen in the HTML generated by the script in example 5.5, and the results were as seen in figure *5.5*. The basic structure here is the *<SELECT>....</SELECT>* tag. As in the example, this typically contains one or more *<option>* tags, which are followed by the values in the selection box. An additional *size=nn* keyword may be included in the *<select>* start tag. This specifies the size of the box in number of lines. If this number is smaller than the number of *<option>* lines, the browser will automatically generate a scroll bar (or possibly a box with page up and page down keys on a character oriented browser) on the display. A value is selected by the user using a graphical workstation's pointer device, such as a mouse. On a character-based display, a value is selected by highlighting the appropriate value and pressing a specific key combination. Likewise, the entered value is returned to the application by activation of a *submit* button, either through the keyboard or through the graphical interface. The application script receives a character string containing the name of the option list (the *name=xxxx* parameter in the *<select>* start tag) and the value following the selected *<option>* tag. Recall from the last chapter that all these variables and their values are passed to the application scripts in a format called URL-encoding, and the application script must parse out these variable names and values.

---

[35]http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html

The select box has other features, including the ability to mark a default selection, to return to the server a value other than the value that appears on the screen and is coded after the *<option>* prompt, and to return multiple selections. This flexibility is available to the application developer with just a few simple commands. Other features are also available. If a select box does not specify a size, for example, but does have more than one *<option>* tag, a graphical browser will generate a drop-down box. Multiple selections are enabled by simply adding the keyword *multiple* to the select tag. A default value (or values, in the case of the *multiple* keyword) is indicated by the *selected* keyword within the *<option>* tag.

The *<input>* tag is one of the most flexible and complicated directives. As we have seen in examples 5.2, 5.5, and 5.7 above, this construct is often used to create a tool which allows the user to activate the application when the input form is filled. The *<INPUT Type=submit>* button is a fixture on almost all interactive forms. The label of a submit button is controlled by the *value=xxxxx* keyword. The *<input>* tag has many other capabilities, however. These functions include the "reset" feature which is an important function on many complicated forms, allowing the user to abort all changes and return to the default values by selecting the appropriate button. Reset is typically coded as follows: *<INPUT TYPE="reset" VALUE="Reset To Default Values">*, although the text for the *value* keyword can be modified as needed.

Another *<input>* option is the *checkbox* type. Coded as *<INPUT Type="checkbox">*, this allows the user to select from a list of options. Any or all of the checkboxes on a form may be checked at any one time. They may share the same name or they may have unique names. The value returned by the browser is the content of the *value* keyword corresponding to the box or boxes checked by the user of the browser. If more than one box is checked, the values are concatenated when returned to the server, joined by a "&". The canonical example for this type is the pizza order form. Here, one might find directives coded as follows:

Example 6.1:

<INPUT TYPE="checkbox" NAME="topping" VALUE="pepperoni"> Pepperoni
<INPUT TYPE="checkbox" NAME="topping" VALUE="onions"> Onions
<INPUT TYPE="checkbox" NAME="topping" VALUE="mushrooms"> Mushrooms

A closely related but orthogonal type is the radio button. This *<input>* type allows selection of "one from many". Only one of a like-named group of buttons may be selected. When one button is pushed in, the others pop out. Care must be taken to insure that radio buttons on a form are named appropriately to ensure that one selection does not inadvertently turn off another selection in another part of the form. This error can happen since the *name* value is the thread that ties all related buttons into a functional system. Accidental reuse of a name can give rise to this condition. This directive might be coded as follows:

Example 6.2:

<INPUT TYPE="radio" NAME="paymethod" VALUE="cash" CHECKED> Cash
<INPUT TYPE="radio" NAME="paymethod" VALUE="check"> Check

Here, the *cash* button is turn on (pushed in) when the form is loaded, as indicated by the *checked* keyword.

Since the contents of a field might be sensitive, HTML also supports a password-style input field. This allows entry of passwords, billing codes, or other information that should not be displayed on a screen. Remember, however, that standard HTTP currently does not support end-to-end encryption, and any passwords or other sensitive information transmitted is subject to scrutiny.[36] When data is keyed into a password field, the display echo is the character "*" (or other fill-in, as deemed appropriate by the browser). A default value may be loaded into the field when a form is first displayed, and this too is shown with the appropriate number of fill characters. This type of input field is coded:

Example 6.3:

<INPUT TYPE="password" NAME="password">

The preloaded value is optionally specified with the *value=xxxxx* keyword, as is typical for the *<input>* directives.

---

[36]Security will be discussed in detail in section 9.3.

The next feature we will mention here is the multiline text box. This is not an *<input>* tag field, using instead the *<TEXTAREA> ..</TEXTAREA>* directives. This tool allows the programmer to define an *n* lines (rows) by *m* (columns) area for text entry. Such a box is often seen on the Web to solicit comments and send mail to the maintainer of a service, but it can be used for anything that requires a larger input area (which the user can see) or better edit support. Recall that a standard input field may contain as many characters as the operator enters, but it will be displayed in a small window (defaulting to 20 characters). The *<textarea>* tag allows easy edit and review of the material entered. This directive might be coded as follows:

Example 6.4:

<TEXTAREA NAME="feedback" ROWS=10 COLS=60></TEXTAREA>

This construct creates a 600 character window that is 10 lines long and 60 characters wide (although the user may enter lines longer than 60 characters). Typically, a graphical browser will support full cursor control and cursor movement keys such as *home* and *end* but wordwrap is not implemented. Characters may be deleted or inserted at any point. When the form is submitted, the data entered is attached to the variable name (*feedback* in this example).

One of the more interesting features of HTML is the *ismap* attribute of the *<IMG>...</IMG>* directive. This tag, combined with the *imagemap* program (for the NCSA server) and its associated configuration file (*imagemap.conf)* allows the developer to create a map, image, icon, or other graphical data representation (such as a pictorial menu) and to have the server respond with links to different URL's in response to pointer selections in discrete portions of the image. One way this is commonly used on the Web is to create a map of an area, a campus for example, and take the user to points of interest based on the user's selection. Another common use is to create a virtual bookshelf icon and allow the user to open books based on the pointer selections on the shelf. The following is an example of this strategy.

First, in the HTML document that is to prompt the user for input, the developer places code such as this:

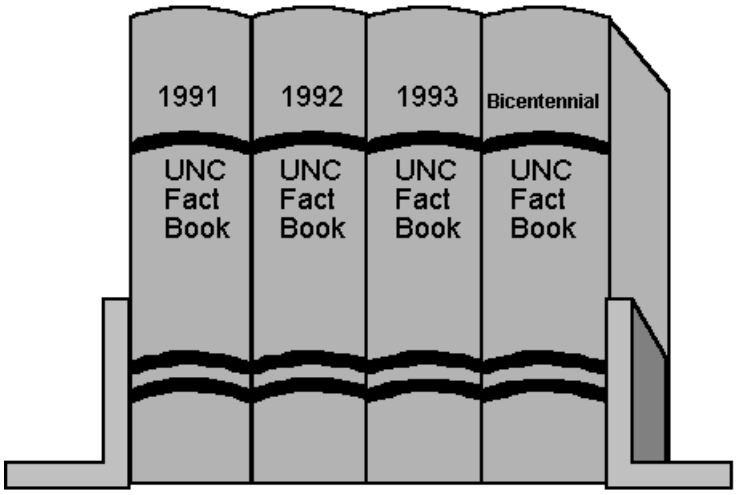
Example 6.5:

```
Click on a book to obtain the information for the specified year.<p>
<A HREF="http://bullhead.adp.unc.edu/cgi-bin/imagemap/books3map">
<IMG SRC="http://bullhead.adp.unc.edu/info/books3.gif" ismap></A>
```

Let's look at example 6.5 one line at a time. The first line simply provides instructions to the user. The second line of this example can be quite confusing, however. This link is a very misleading URL (and in fact is not a real URL), since there is no *..cgi-bin/imagemap/books3map* HTML document on the server *www.adp.unc.edu*. The executable script in the CGI library is actually *imagemap*. The trailing information (books3map) is actually a parameter that is passed to the input processing program, *imagemap*. The third and final line is a directive that loads the image into the browser to provide a context for the (x,y) coordinate selection. The *ismap* option after the file name tells the browser that the link (in line 2) is to be processed as a map and not as a pure link.

What has happened here is that a graphic such as the following is displayed. Figure 6.1 below is the image in the file *books3.gif*.

Figure 6.1:



To make this image generate different URL's for different selection positions, the developer must create a configuration file. This is the file *imagemap.conf*, as mentioned above. It will have an entry for each image that supports coordinate mapping. The line that defines our system is as follows.

Example 6.6:

```
books3map:/home/www/htdocs/info/books3.map:
```

The file */home/www/htdocs/info/books3.map* contains the coordinate mapping in the image. This coordinate mapping is typically expressed as a rectangle or circle, but other forms are possible.[37] Like many features on the Web, documentation and source code for the *imagemap* program may be found on the NCSA Web server.[38] The detailed mapping for this example (which uses only rectangular coordinates, listed by pixel location) is as follows.

Example 6.7:

```
default http://bullhead.adp.unc.edu/info/none.html
rect gopher://bullhead.adp.unc.edu:70/1/unc.fact.book/unc.fact.book.1991 79,1,154,299
rect gopher://bullhead.adp.unc.edu:70/1/unc.fact.book/unc.fact.book.1992 155,1,226,299
rect http://bullhead.adp.unc.edu/info/fb94/fctbk94.html 227,1 297,299
rect gopher://bullhead.adp.unc.edu:70/1/unc.fact.book/unc.fact.book.bicent 298,1,412,299
```

---

[37]Alternate location identification strategies include polygons (of up to 100 vertices) and the "nearest point" method, in which the shortest distance from the click to a defined point determines the URL.
[38]http://hoohoo.ncsa.uiuc.edu/docs/setup/admin/Imagemap.html

This allows the server to link the browser to a distinct URL for each of the four books on the shelf. The reader can see that three of these URL's are references to a Gopher server; one is to another HTML page. The first line of example 6.7 defines the action to be taken if the user selects an area that has not been defined in a region associated with a specific URL.

The imagemap facility is a highly useful, if somewhat cumbersome, method to extend the data input tools available in HTML. For example, this can be used such novel applications as on-line frog dissection[39] to allow a student to see if the he or she understands where to make cuts during a dissection procedure. This particular application will be discussed further in section 10.4. It is likely that the imagemap process will become more streamlined and easier to use as HTML servers continue to mature. There are already tools available that aid the user in creation of the coordinate-to-URL mapping, such as is shown in example 6.7. In the Microsoft Windows environment, for example, the program *mapedit*[40] is a tool that can dramatically ease this process. A version of this program is also available for various UNIX-based operating systems, as well.

[39]http://curry.edschool.Virginia.EDU/~insttech/frog/
[40]ftp://sunsite.unc.edu/pub/packages/infosystems/tools/mapedit/mapedit.zip

6.2 - How is an application accessed?

Once an application has been created, it is invoked by a browser that requests its initial URL. When a browser requests a static document, the server daemon reads the requested file and passes the stream directly to the browser. If the URL requested is resident in the CGI directory, however, it is processed differently. First, a script file is executed under the daemon's operating system. The output of this script becomes the HTML that defines the application page.

If a browser's display is generated with HTML directly retrieved from the server's document directory, it is available for perusal (and in fact may be saved and modified for use on other servers). There is nothing in the Web model that prevents this dissemination of source code. If a page is generated by a CGI script, however, the generated HTML may be saved but the original CGI script may not be retrieved in this fashion. The reader may not consider this to be a major issue, but it is worth noting that the secrets of HTML page design (but not CGI script design) can be learned by simply reviewing existing work. Also, this may become an issue if at some point commercial application systems do become commonplace on the Web.

The HTML that is loaded into the browser is not syntax checked by the server. It is not syntax checked when it is loaded into the server's document directory. It is quite easy to create HTML that will not only fail to display properly but which will, in fact, corrupt and crash a browser. CGI scripts are checked by their respective interpreters or compilers as they are processed (or when they are compiled), but the error message from the script interpreters may appear embedded in the stream of HTML from the server, often resulting in quite unpredictable results. Needless to say, the HTML generated by a CGI script is not validated!

Chapter 7 - Browser Differences

7.1 - Application Appearance on Diverse Clients

In Section 3.2, we briefly discussed some of the diverse clients that appear on the Web. These browsers are often written for different operating environments and different operating systems. To further this point, presented here are formatted screens from some sample browsers. The application used here is the "who/finger" application explained in example 5.4 and shown as a formatted screen (from *Mosaic* for Microsoft Windows) in figure *5.5*.

The lowest common denominator for forms-based application development on the Web is probably the character mode browser, *LYNX*. *LYNX* is available to any user of a UNIX system with access to a VT100 (or similar) terminal or emulator. Basically, it can be accessed by virtually anyone in the world with a computer and a communications link. Public *LYNX* clients are available worldwide[41].

Here is the application, as displayed in *LYNX*:

Figure 7.1.1:

```
                                                    Who's Home? (and finger them)

                          WHO'S HOME? (AND FINGER THEM)

   This will show who's home at Bullhead!


   ( ) sybase
   ( ) longlegs

   To finger someone, press this button: Finger.




(Checkbox Field)   Use right-arrow or <return> to toggle.
  Arrow keys: Up and Down to move. Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Keyboard actions simulate the action of a pointing device in the LYNX environment.

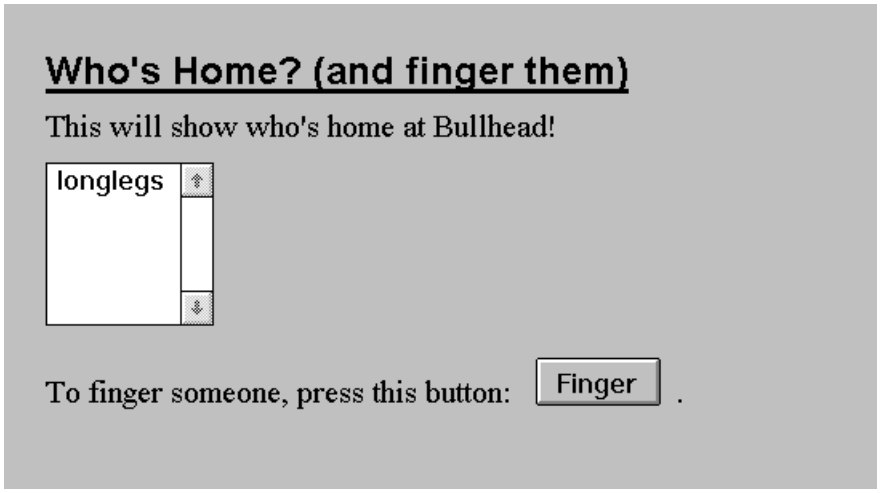
---

[41]One such system offering this service is the Sunsite, located at UNC-Chapel Hill. Telnet to "sunsite.unc.edu" and log in as "lynx" to access this program.

Even on the same operating system (or operating environment), browsers may have a substantially different look and feel. Here below is a copy of the same screen shown in figure *5.5*, but here displayed on the browser *WinWeb*, a browser for Microsoft Windows.

As compared to figure *5.5*, note the inactive (light gray) scroll bar. The designers of *Mosaic* chose to implement the case of a non-scrolling display by not showing the scroll bar at all. Note that this has implications for user documentation and overall "look and feel".

Figure 7.1.2:



The browser *Cello,* for example, cannot display forms in its current release[42]. Although *WinWeb* can handle many forms, it currently fails to display multiline text boxes. Even as new releases of these and other Web browsers make their appearance in commercial offerings or on FTP sites, there will be a significant transition period during which the capabilities of browsers will be suspect.

[42]There is a new release of this browser under development that will include forms processing. This should be available during the first quarter of 1995.

7.2 - Feature Set of Target Browsers

One problem with application development that may be apparent from the discussions so far is the determination of the minimal feature required of the browsers to process an application. If an application developer has a captive audience and can dictate the choice of browsers, then there is unlikely to be a problem with unexpected or erroneous behavior since the developer can test all forms and pages in an application with the target browser. However, this is highly unlikely to be the case, since the heterogeneous nature of the Web environment encourages end users to select browsers that cater to their tastes. If a target application environment is isolated enough to permit browser specification, it is likely that other proprietary (and more mature) development tools would offer a better choice for code development. Indeed, the strength that HTML applications bring is based on their diversity.

As mentioned above, the simplest target for application development might be considered to be the UNIX character browser, *LYNX*. However, even though it does not have a graphical interface capable of supporting image maps, for example, it is otherwise full featured, supporting most forms-based constructs. This leads to an interesting conundrum -- develop for a platform that requires users to emulate a dumb terminal on a timesharing machine, or accept the limitations in forms support evident in many personal workstation browsers. The latter choice is likely to be the path most often taken, since another of the strengths in HTML applications is the capability to offer a multimedia presentation of an application. Also, the rapid pace of browser development is likely to ensure that deficiencies are addressed so that use of expanded forms constructs will be permitted.

Chapter 8 - Data Access Tools

Data manipulation is the heart of most application systems. HTTP servers offer access to data resources through scripting facilities. Early servers did not offer this feature, but capabilities of servers in forms and script processing have proceeded rapidly[43]. As discussed in Chapter 5, the basic model of data manipulation for the CERN and NCSA HTTP servers is the CGI script. Extensive examples of this process may be found on the Web, including the *GSQL* system from NCSA[44]. With the background developed so far, we will now turn to a more detailed explanation of the tools used to create applications that access generic database resources as well as provide a further explanation of the *GSQL* system.

GSQL consists of two primary programs, *gsql* and *sqlmain*. The first of these routines, *gsql*, allows a programmer to define a form with its own somewhat cumbersome scripting language. The second routine, *sqlmain*, performs the interface with an SQL database. The examples in the *GSQL* distribution are specific to *SYBASE*, but instructions are included on creating interfaces for other databases such as *Oracle*. The programs in *GSQL*, however, do not perform any processes that are technically difficult. These programs may be best viewed as an example or starting place to allow a programmer to see working SQL code before attempting to modify the interface to work in a particular server environment. Recall that *gsql* has two parts -- forms generation and database access. It may be more straightforward to create forms directly without using the forms procedure script language of *GSQL*, especially if the developer employs a full-featured HTML editor. To make this point clear, let's repeat the examples from Chapter 5 (5.7-5.9), but this time specifically look at the *gsql* interface.

---

[43]http://info.cern.ch/hypertext/WWW/Daemon/Features.html and
http://hoohoo.ncsa.uiuc.edu/docs/info/VersionHist.html
[44]http://www.ncsa.uiuc.edu/SDG/People/jason/pub/gsql/starthere.html

Example 8.1:

```
#!/bin/sh
#
# Joel Dunn - Last modified October 1994

echo Content-type: text/html
echo

cat << EOM
<TITLE>Who's in the Email Data Base?</TITLE>
<H1>Who's in the Email Data Base?</H1>

<FORM METHOD="POST" ACTION="http://www.adp.unc.edu/cgi-bin/unc_number.sh">

Enter your UNC Number ---> <inPUT NAME="unc_numb">
<P>
<inPUT TYPE="submit" VALUE="Look Up">
</FORM>
EOM
```

Example 8.2:

```
#!/bin/sh
#
# Joel Dunn - Last modified October 1994

echo Content-type: text/html
echo

eval `/home/www/cgi-bin/cgiparse.run $CONTENT_LENGTH`

        cat << EOM
<H1>SQL Query Output</H1>
<PRE>
EOM
/home/www/cgi-bin/unc_number.sh2 $unc_numb
echo \</PRE\>
```

Example 8.3:

```
#!/bin/sh
/home/www/gsql/sqlmain "SELECT amdbemail_t..unc_people.last_name FROM \
amdbemail_t..unc_people WHERE amdbemail_t..unc_people.unc_id = '$1'" \
"LOGIN my_userid" \
"PASSWORD my_password" \
"SYBASE_ENV SYBASE=data_path" \
"SYBASE_ENV2 DSQUERY=server_name"
```

The form is constructed by execution of a standard shell script in example 8.1 (it does not use the *gsql* program), but the query in example 8.3 does call *sqlmain* and passes through the *SYBASE*-specific routines that are a part of the *gsql* package's distribution. The database processing in example 8.3 could have been done with any program that takes query information and interfaces with the SQL database through the database's interface library. There is nothing in *sqlmain* that is required for SQL access in the Web environment.

The NCSA server pages for *GSQL* do contain hyperlinks to working database query applications that may be used as models for further work. One such example is the NCSA help desk[45], which incidentally also combines many elements of fill-in forms. This example allows a user to query the problem database of NCSA according to criteria entered *ad hoc* or selected from pre-existing lists. The Agency for Toxic Substances and Disease Registry, which can be referenced from the *GSQL* pages, also has an informative inquiry program that not only performs database queries but also dynamically generates "clickable" hyperlinks to other parts of its database[46].

The CGI script model can be used to access any database that is local to the server or is on an attached network. It does not have to use SQL. The database merely needs to provide library access routines or utilities that will perform queries and return output to the *stdout* file or the equivalent console device if the server is running on a non-UNIX operating system. These database routines do not need to understand or even know of the existence of CGI and HTML. The wrapper programs that invoke the database access routines may place all necessary prolog and epilog HTML directives in the output stream.

The CGI interface script facilities are intimately tied to the operating system of the server. There is no CGI language, *per se*. In UNIX, these scripts may be written for the Bourne Shell, the Korn Shell, the C Shell or any installed shell, or could in fact be for another interpreted language such as PERL or a compiled language such as C. Forms variables are presented through input redirection functions and must be parsed and broken apart for script processing.

As an example for this type of system, consider access to a WAIS database. While it is true that current versions of *Xmosaic* contain library routines to directly access WAIS servers, most browsers still do not incorporate direct WAIS connections. This deficiency may be alleviated by using a properly configured

---

[45]http://base.ncsa.uiuc.edu:1234/htbin/hmenu
[46]http://atsdr1.atsdr.cdc.gov:8080/hazdat.html

Gopher server as an intermediary, but this is also an ideal opportunity to illustrate the flexibility of the CGI interface. Additionally, this example shows the multiple language flexibility of a CGI system, starting with a Bourne shell script and passing control to PERL routines. Here are three scripts that may be used to illustrate this point, demonstrating access to a WAIS database of HTML forms on a server. This database contains an index to the text of each HTML document on the server *www.adp.unc.edu*. The first of these scripts is *wais-html-q.sh*, shown here in example 8.4. Control is passed to this script from a form, as in examples 8.1 and 8.2.

Example 8.4:

```
#!/bin/sh
eval `/home/www/cgi-bin/cgiparse.run $CONTENT_LENGTH`
keywords=`echo $keywords | sed -e "s/ /%20/g"`
/home/www/cgi-bin/wais-html.pl $keywords
```

The script *wais-html-q.sh* parses the input string, manipulates the parameter through the UNIX stream editor (sed), and calls the PERL script *wais-html.pl* (shown here in example 8.5) with the search keyword that was extracted from the input string. This is where most processing occurs. This script calls the actual WAIS engine to retrieve index information from the database and does the bulk of the work in reformatting that output into a usable HTML stream.

Example 8.5:

```
#!/bin/perl
# wais-html.pl -- WAIS search on html archives
#
# Joel Dunn - August 1994
#   based on:
# Tony Sanders <sanders@bsdi.com>, Nov 1993, and marca@ncsa.uiuc.edu


$waisq = "/home/wais/wais-8-b5.1/bin/waisq";
$waisd = "/home/longlegs/wais-sources";

sub do_wais {
   $query = $ARGV[0];
   $pquery = $query;
   $pquery =~ s/%20/ /g;

   open(WAISQ, "-|") || exec ($waisq, "-c", $waisd,
            "-f", "-", "-S", "htdocs.src", "-g", $pquery);
   print "Content-type: text/html\n\n";
   $title = "WAIS Search Engine";
   print "<HEAD>\n<TITLE>", $title, "</TITLE>\n</HEAD>\n";
   print "<BODY>\n<H1>", $title, "</H1>\n";
```

```
    print "The html archives contain the following\n";
    print "items relevant to \'$pquery\':<P>\n";
    print "<DL>\n";

    local($hits, $score, $headline, $lines, $bytes, $type, $date, $id, $id2);
    while (<WAISQ>) {
        /:score\s+(\d+)/ && ($score = $1);
        /:number-of-lines\s+(\d+)/ && ($lines = $1);
        /:number-of-bytes\s+(\d+)/ && ($bytes = $1);
        /:type "(.*)"/ && ($type = $1);
        /:headline "(.*)"/ && ($headline = $1);
                /:original-local-id\s+(.*)/ && ($id = $1, &anytostring);
        /:date "(\d+)"/ && ($date = $1, $hits++, &docdone);
    }
    close(WAISQ);
    print "</DL>\n";

    if ($hits == 0) {
        print "Nothing found.\n";
    }
    print "</BODY>\n";
}

sub docdone {
    if ($headline =~ /Search produced no result/) {
        print "Sorry, no search results.<P>\n";
    } else {
        $headline =~ /(.*)\s+/ && ($headline = $1);
        print "<DT><A HREF=http://www.adp.unc.edu/cgi-bin/";
        print "waisretrieve.pl?$id2:htdocs>$headline</A>\n";
        print "<DD><b>Score:</b> $score, <b>Lines:</b> $lines, ";
        print "<b>Bytes:</b> $bytes\n";
    }
    $score = $headline = $lines = $bytes = $type = $date = '';
}

sub anytostring {
    local($anysize,$numvals,$evalpgm);
    $id =~ /:size\s+(\d+)/ && ($anysize = $1);
    $id =~ /#\(+\s+([\s0-9]*)/ && ($numvals = $1);
    $numvals =~ s/\s+/,/g;
    local($i) = 0;
    local($j);
    $id2 = "";
    for (;$i < $anysize; $i++)
                {
                $numvals =~ s/\s?([0-9]*),/$1a/ && ($j = pack("c",$1));
                $id2 = $id2 . $j;
                }
    $id2 =~ s/\s+/%20/g;
}

eval '&do_wais';
```

In this example, there are three main sections, the PERL subroutines[47] *do_wais*, *docdone*, and *anytostring*. The script begins by invoking *do_wais* in the last line of the example. This code first executes the WAIS query and pipes the output from this step into a file. The script then outputs HTML prolog information with the *<head>...</head>* and *<title>...</title>* directives. Each query output record is

[47]A PERL subroutine is denoted by the keyword "sub" preceding the subroutine name.

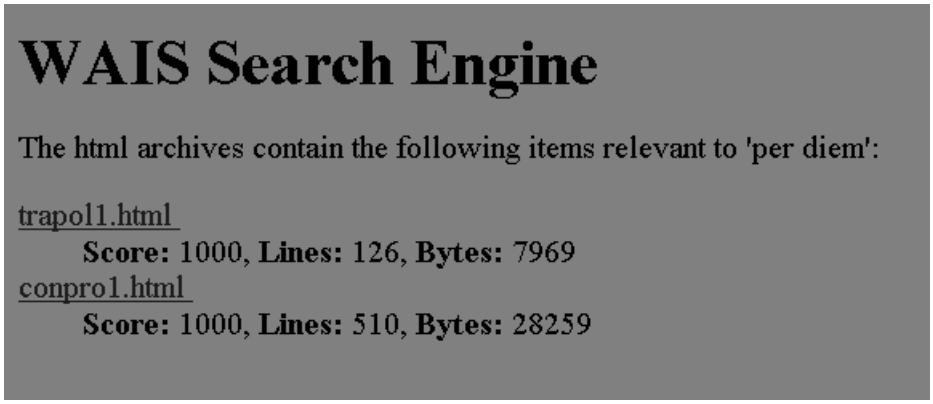
parsed (*anytostring*) and finally output (*docdone*) with the name of another script and a formatted parameter string[48] as the new URL in the hypertext anchor area (the area between the <a href> and directives; see example 5.1). When the query lines have all been processed, the PERL script outputs the HTML epilog from the original *do_wais* subroutine. The output shown below is an example of the results of a query. The document titles, shown with underlines here in figure 8.1, are the hyperlinks that invoke the *waisretrieve.pl* script (in example 8.6) to actually retrieve each document. Note how the variables *score*, *lines*, and *bytes*[49] have been inserted for each document, and note that the headings have been emphasized in bold for readability. The query that was used to generate the output in figure 8.1 was the two word string *per diem*, and was used to search the business manual of the University for relevant information on travel policies and reimbursement. The headline information here, the document title, is simply the name the name of the HTML file that contains the reference. This is an artifact of the WAIS software used to index the pages. This version of WAIS predated widespread use of HTML, and consequently did not have indexing support to provide easily readable headlines.

---

[48]The results of a WAIS query are returned in a form called *any*, a generic machine independent format that is portable among WAIS implementations but which is not a representation that lends itself to passing as a script parameter (many imbedded spaces, for example). The routine *anytostring* converts the relevant information to a simple string which can be used as a parameter.

[49]These variables are returned by the WAIS query. The score is the weighted value of the hit as computed by the WAIS search engine. The variables *lines* and *bytes* describe the size of the document that will be retrieved if this link is selected.

Figure 8.1:



The model described in example 8.5 (used to create an HTML page) is also seen in the output of the subsequent script, *waisretrieve.pl*, shown in example 8.6. This script takes the string-formatted input parameter and retrieves a document from the WAIS archive using the routine *waisretrieve*. *Waisretrieve* is simply another Bourne shell script that itself invokes *waisq*. The output of *waisretrieve* is piped into a PERL file so that this output may be rewritten to standard output according to CGI standards. *Waisretrieve.pl* then itself writes the document to standard output, where it becomes sandwiched between HTML tags. Due to the smaller size of this script and its simpler task, the reader may find this example easier to follow. The same principles found in this script, however, are applicable to example 8.5. Note that the use of the HTML directives and context descriptions in this script mean that the information retrieved from the WAIS archive could be plain text [50] or it could actually be an HTML document itself, as is the case in this example.

Example 8.6:

```
#!/bin/perl
# waisretrieve - retrieve a document from a WAIS archive
#
# Joel Dunn - August 1994

$waisr = "/home/wais/wais-8-b5.1/bin/waisretrieve";

sub do_wais {
   $doc = $ARGV[0];
   $pdoc = $doc;
   $pdoc =~ s/%20/ /g;

   print "Content-type: text/html\n\n";
```

[50] See http://www.adp.unc.edu/cgi-bin/wais-flyfish-q for an example of how these same scripts are used to retrieve the indexed messages from the flyfishing listserver.

```
    $title = "WAIS Retrieval Engine";
    print "<HEAD>\n<TITLE>", $title, "</TITLE>\n</HEAD>\n";
    print "<BODY>\n<H1>", $title, "</H1>\n";

    print "The document \'$pdoc'\ is:\n";
    print "<P>\n";
    print "<PRE>\n";

    open(WAISQ, "-|") || exec ($waisr, "$pdoc");

    while (<WAISQ>) {
        print;
    }
    close(WAISQ);
    print "</PRE>\n";
    print "</BODY>\n";
}

eval '&do_wais';
```

The discussions presented so far in this chapter have focused on data retrieval. There is, of course, another facet to this issue, the area of database modification. This is an area where the current tools and development models are less adequate. This problem and others will be exposed further in the following chapter. Section 9.4 covers specific issues in database transaction management.

Chapter 9 - Further Problems in Application Development

9.1 - Language Issues

HTML is an emerging standard[51]. As more developers bring (and improve) browsers and servers to the World Wide Web, HTML application developers are continually creating innovative systems and are pushing the bounds of the language. The standard definition of HTML (Level 1) at first did not include forms for data entry. However, this is still the minimum level of compatibility for browsers. Specification for user forms was addressed in what is known as HTML Level 2. There is work underway that will yield the successor to HTML 2, variously known as HTML 3 or HTML+. This specification may or may not be a proper superset of HTML 2. It includes such features as tables, figures, and mathematical equations.

There is considerable variability in the way browsers respond to different levels of language usage. Some early generation browsers that do not support forms merely display what they can and ignore unknown directives. The application is unusable, but typically no harm is done. What is unfortunately typical, however, is that often misuse of directives or combinations that are technically invalid may simply cause a browser to freeze, or "lock up". This can have potentially damaging results depending on the type of application that is being accessed. The problem is further compounded since the most widespread browser family (*Mosaic*, on various platforms) is very tolerant with "creative" uses of HTML. It is not at all uncommon for frequently accessed HTML pages or applications that work properly with *Mosaic* to fail when accessed with another browser. What is needed, perhaps, is a validation suite that can be applied to new browsers to determine their compatibility, especially in the area of forms processing.

In addition to technical standards, there are many stylistic standards[52]. These are focused primary at general design of HTML pages, but the points raised are often valid for interactive applications, as well.

---

[51]http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html
[52]http://info.cern.ch/hypertext/WWW/Provider/Style/Introduction.html and
http://www.willamette.edu/html-composition/strict-html.html, for example

For example, an application will often combine hypertext prose with interactive forms. The hypertext should be well constructed and not burdened with "click here" links[53]. The text should stand on its own at least as an outline of the ideas explored through active links. Page size and multimedia object size should be evaluated with respect to the bandwidth requirements for effective utilization of the pages. The lessons learned in study of traditional user interfaces should not be forgotten, since the initial pleasure in exposure to multimedia presentations will quickly fade if the application is poorly designed, or if gadgets overwhelm useful information.

[53]A "click here" link is one that might say, "For more information on HTML, click here" where the words "click here" are the anchor for a hypertext link. It is considered better style to write naturally and to provide a link with the context of a more content-rich sentence. This might be "HTML is a hypertext markup language", where the acronym HTML is the hypertext anchor, for example.

9.2 - Statefulness and Recovery

When we speak of *state* in a computer system, we mean information that either or both parties in a conversation may retain so that they can ascertain which of several intermediate stages in a process describes the status of the principals. State information in a client/server system can be kept in either the client or in the server, or in both. This information can be useful to help a server optimize performance, for example, by caching reusable information for later reuse. A file server might retain file pointers to minimize the amount of information that a client has to send to the server on each request. Alternatively, a client of a file server can store its own file pointers and send exact specifications to the server for each request. Each method can be useful, but there are tradeoffs. If the client is *stateful*, it may make the system less efficient by requiring larger messages or recalculations in the server. If the server is *stateful*, it may complicate recovery if a single client fails or group of clients fail at once.

When we consider state in an HTML application, we must look at the environment in which it operates. An HTML application system is likely to function across the Internet and be subject to many possible communication breakdowns. Should state be retained, and if so, which party in the conversation should retain the state?

In a purely stateless system (where both the client and server are stateless), error recovery is a significant challenge. Each exchange between an HTTP server and its client can be considered an atomic transaction.[54] Each client's request spawns an instance of the server (or utilizes a thread, depending on the implementation) to process that action. An HTTP daemon is a concurrent, connection-oriented server [Comer, Stevens 1993]. All requests during a particular transaction are between the client and one particular instance of the server[55]. Any propagation of state from one transaction to the next must be completely done with information stored at the server, or information passed to the browser with each

---

[54]See *figure 3.2.1* for a simplistic diagram of a client server exchange.
[55]A transaction is defined here to be the information passed from the client to server and back in response to the request by a client for a particular URL.

response. Remember that the browser is simply a tool to create display items such as fields and list boxes, and to then accept user input. How then *can* it retain any state information, if this is indeed desirable?

One way to accomplish this is for the server to generate URL's as links in dynamically created HTML pages (by creating temporary file names and referring to those names in the URL). These URL's not only tell the server upon their receipt what information the client might need but in fact can carry information about the client, such as the stage of a transaction that has been started by a client. This is much like the concept of the *nonce*[56] in some authentication schemes [Burrows, et. al., 1990]. If a particular URL is returned by the client, then that client must be the specific one that had a quantifiable communication with the server at some earlier time. This allows the server to then store whatever state information is desired about the session and the client. However, this places the burden of the identification effort on the server (thus creating a stateful server), since the URL must be a construct (such as a file) that is recognized by the server. An interesting example of this can be seen in the payment system developed by OpenMarket.[57] The state information in this scheme includes the concept of a *payment URL*[58], which encodes details of what is for sale and the price at which it is offered. This may be processed by a payment server (performing authentication, to be discussed later), which then issues an *access URL*, encoding information about what was purchased, and at what price. It is described by OpenMarket as a "paid digital invoice".

Another method to make this system stateful is to build the state information into the HTML page processed by the browser. Various techniques, such as input fields with zero lengths, have been used in the past but one difficulty with such efforts is that this stretching of the application development language often leads to different results on different browsers. A construct that has a benign impact on one display may adversely impact presentation on another since there are no standards for the way such misuse of the

---

[56]A *nonce* is a token generated by one party in a conversation to hand to another party. If that nonce is subsequently returned, it carries implicit information that the second party has been in the conversation before, and may in fact provide information about the stage reached by parties in a conversation.
[57]http://www.openmarket.com/about/technical/payment/overview.html
[58]http://www.openmarket.com/about/technical/payment/urls.html

language is handled. Currently, the preferred method for loading information into a client for eventual (unaltered) return is the *TYPE="hidden"* keyword of the <*INPUT*> HTML directive. When used with the *VALUE* keyword, this syntax allows information to be sent from the server to the client as the argument to *VALUE,* with the knowledge that the client program has not offered the user an opportunity to change this value. However, this construct is unfortunately not fully supported on all browsers, and may itself cause unexpected problems. Additionally, this keyword allows information to be stored only for the duration of the single page. To retain information for a longer period (over an application system of several forms, perhaps), the developer has to continually resend the directive (and data) with each subsequent form.

To illustrate the independence of the transaction partners, the network link between the partners may actually be broken and re-established some significant time later without impacting the connection.[59] Obviously, this could complicate any interactive data entry application! A request made by the client might never be seen by the server, and the server has no way of even knowing that the client intended to make a further request. Since the client is a reactive program, simply rendering HTML and returning data, the application logic actually resides in the CGI scripts of the HTTP server, and the client does not know that the transaction was lost. The only indication that something might be amiss is that the human user of the client does not see an anticipated response. Depending on operator observation to ensure system integrity is a very ineffective way to provide auditable control.

Is it desirable to store state information? To aid in the transition of HTML to a full-fledged development language, it will be necessary to formalize storage of *some* form of state information at the client. This will remove the burden of forcing state implementations onto the server by techniques such as encoding information into the specification of the URL. Excessive state information at the server does indeed pose its own recovery problem in a real network which is subject to the myriad problems that can impact network reliability. An HTML browser, however, in its role as a reactive generator of display

[59]The author uses a SLIP connection to the Internet from his home office, and the phone link may actually be put "on hook" and the line used for other purposes, and then re-established without impacting the apparent status of the HTTP client. A similarly connected Telnet or FTP session would be terminated.

controls, is different from a client constructed for a specific task from a traditional third or fourth generation programming language. The traditional client has extensive facilities to store whatever information is deemed appropriate by the client designer. A programmer who seeks to build an application with HTML currently does not have access to local storage. Some modest, fixed amount of local heap space (accessible via a new HTML directive, perhaps[60]) would greatly improve the ability of an application designer to create robust application systems.

The idea of using the Web as a front-end for database applications is commonly expressed in the Internet community today. There are many projects underway to explore this possibility. The *Genera* project[61] at Yale University provides yet another look at HTML language extensions to which add functionality by expanding the types of database applications which may be developed. These extensions include such items as spreadsheet display objects and a generalized store/retrieve directive to allow the client to have a finer-grained control over local events.[62]

Application systems using the Web are becoming commonplace. However, it is still beneficial for the user of such systems to understand the flow of information and the places where smooth functionality can be disrupted. Certain types of applications may be suitable for use in an emerging application development environment. However, other mission critical systems or systems with large potential costs of failure may, for now, be better served by other mediums with proven track records for robustness.

---

[60]For example, this might be encoded <STORE VarName=xxxxxx,Value=yyyyyy>, with the corollary <RETRIEVE VarName=xxxxxxx>. Storing a null value could free the resource. The scope of this could be global for the life of the client, or have some other delimiter. Separating this function from the <INPUT> directive (with the TYPE="hidden" keyword) would allow the application developer more flexibility in specifying the way such information is retained.

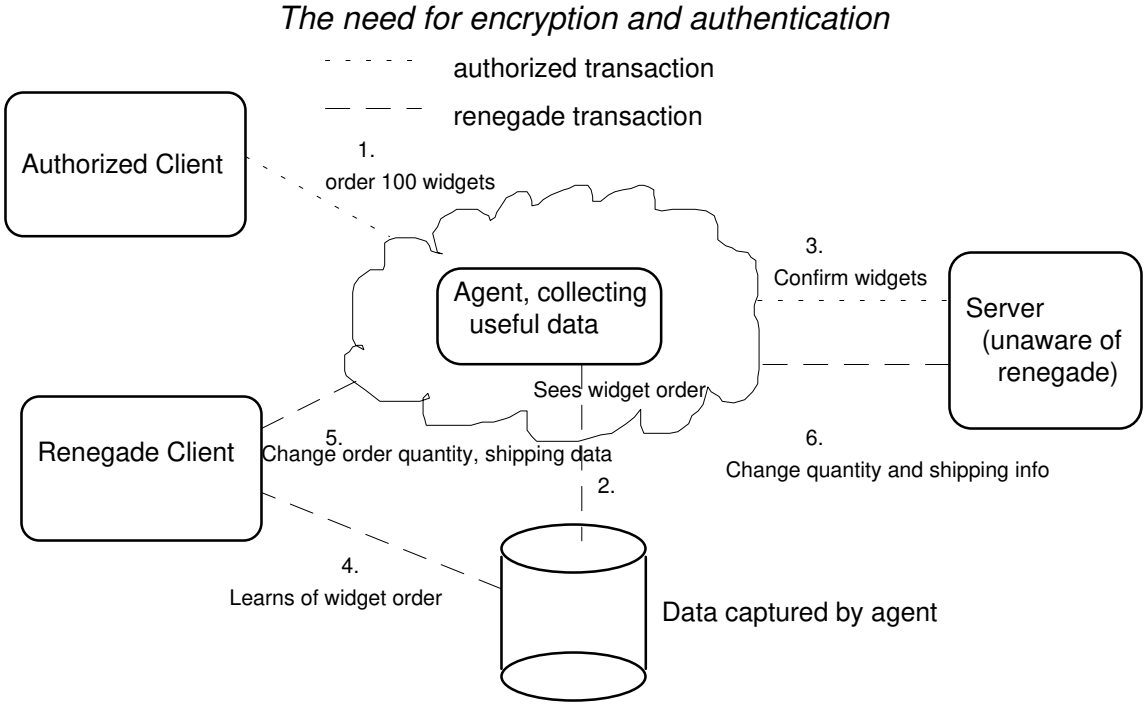
[61]http://gdbdoc.gdb.org/letovsky/genera/genera.html

[62]http://gdbdoc.gdb.org/letovsky/genera/genfuture.html

9.3 - Security

Security in open networks is a complex subject. There are many potential weaknesses in distributed systems, and many diverse types of attacks. The reader interested in detailed background on these issues can consult Voydock and Kent, "Security Mechanisms in High-Level Network Protocols" [Voydock, 1983]. To better understand the security issues specifically inherent in the design of current HTTP clients and servers, it is advantageous to consider the flow of a hypothetical transaction such as the following:

Figure 9.3.1:



In this example, an authorized client makes an on-line order of 100 widgets[63] (step 1). Since this is an unencrypted transaction, a monitoring agent in the internet sees the order and captures this data (step 2). In the meantime, the server sees the order for 100 widgets and initiates the order process, charging the bank account of the authorized client. The renegade client, operating in conjunction with the monitoring agent, is notified of the widget order (step 4). Spoofing as the authorized client, the renegade orders a

[63]The use of the term *widget* is strictly in the sense of economic analysis where an economist may define some tangible good that is offered for commerce. This use of the term has no correlation to any meanings the term may hold in computer science.

change in the order -- the quantity is changed to 105 widgets, with instructions to drop off the extra 5 widgets at an alternative address, directing them to a site controlled by the renegade (step 5). Since any passwords used by the authorized client were not encrypted and thus subject to analysis, this is a simple task. The server sees the request (step 6) and changes the order information accordingly, since there is no signature on the request and it assumes that from the knowledge of the transaction (e.g. order number, confirmation number, etc. that is in the possession of both the authorized client and the renegade) that the request is valid. Consequently, the renegade has just stolen 5 widgets. With luck for the renegade, this may not be noticed for a significant time, if at all. The customer got 100 widgets; although the bill the real customer received was 5% higher than expected, that might be explained by fluctuations in the widget market, so the bill is paid. Obviously, strong accounting controls at the customer's site could catch such a transaction, but it may be too late to actually catch the perpetrator or recover the stolen widgets.

What can be done about the theft just described? This is an artificial example, but it does serve to illustrate the basic security problems with this model. There are really two problems here -- encryption (transport security) and authentication (identity validation). Traditional shared-key encryption solves only a part of the problem (transport security) unless you assume that the shared key cannot have fallen into the wrong hands. If you assume that the key can only be in the hands of a legitimate user, then only a legitimate user could have sent the message [Burrows, et. al., 1990]. This model may not be appropriate for all types of commercial transactions envisioned for the World Wide Web. This would require an exchange of shared keys between customer and vendor prior to transaction initiation. A better methodology would be one that allows spontaneous, secure communications between interested parties. Some form of public key encryption is likely to be more appropriate.

Public key technology[64] involves the use of a pair of related keys, one of which is freely distributed and one of which is tightly controlled by the owner. These keys are algorithmically related such

[64]For a broad background on encryption, especially public encryption and its application to the network community, see http://www.rsa.com/faq/faq_toc.html

that a message encrypted by user **A** with the public key of user **B** can only be decrypted by user **B**, even though the encryption key used by **A** is known to all. Furthermore, public keys have the property that if a message is encrypted with user **A**'s secret key, then user **B**, in possession of user **A**'s freely distributed public key, can decrypt the "signature" and authenticate that only user **A** could have originated the message. This has obvious benefits for a distributed system: clients and servers could use publicly known keys and encrypt messages or decrypt signatures to both protect and validate transactions. There is, however, a major problem...dissemination of public keys. Although one part of the key pair is public, there is nothing inherent in the model to prevent user **C** from placing a public key in the public domain and claiming it to be **A**'s public key. If it is accepted by the community as **A**'s public key, then messages encrypted with the key can be read by **C**, and messages signed by **C** will be accepted as originating with **A**. This invokes the problem of the trusted "key registry", or key certification. Who is ultimately trusted to keep and disseminate public keys?

Again, this key registry model might not be the most conducive for the kind of open connectivity that has so far characterized the World Wide Web. The security and authentication needs of the Web exist for casual, spontaneous sessions as well as preplanned, registered sessions. So far, non-proprietary schemes for secure information transmission require the end user to have and implement a free-standing version of a security program such as PGP (Pretty Good Privacy) or RIPEM.[65] This is typically implemented on the UNIX platform only (defeating the heterogeneity of the Web) using the finger service between the client and the server to retrieve the public keys. In the NCSA implementation, the server uses a local public key file from which a pre-registered key is retrieved using a name sent to the server by the client. Both PGP and RIPEM are supported in NCSA's server, for example, but these products require RSA support in the encoder/decoder and may create licensing or export problems due to governmental controls on cryptography technology[66]. Problems with this implementation are the exchange of keys, lack of

---

[65]http://hoohoo.ncsa.uiuc.edu/docs/PEMPGP.html

[66]For information about licensing and export controls, again refer to http://www.rsa.com/faq/faq_toc.html as a starting point for further data.

spontaneity (requiring pre-registration), and the lack of timestamps on messages [Smith, Tygar 1994], permitting a malicious attack through message replay.

There is considerable work underway to implement a general solution to the issues of encryption and authentication. In one such approach, a consortium of organizations led by Enterprise Integration Technologies (EIT) is currently working on creating a revision to the HTTP protocol, to be called Secure HTTP (S-HTTP). A working draft of this specification exists at this writing, but it does not yet have the status of a formal RFC[67]. The intent of this effort is to produce a flexible yet powerful specification to allow development of interoperable secure clients and servers, yet not to mandate any particular algorithms or cryptographic systems. This revised protocol is to be a superset of HTTP, enabling old clients to exchange unsecured information with a secure server, or permitting a secure client to retrieve unsecured information from a pre-S-HTTP server in the way that most of the Web operates today.

Recall that the model of communication of the Web is one that assumes that the client will take a stream of data from the server and render it into display instances.[68] Under the current HTTP implementations, the client does not know that this is a request for a secured transaction until it attempts to contact the server. This means that with the present security schemes, the first communication of a secured transaction is in clear text until it reaches the server and is denied. Only then does the client realize that it must use secure techniques to communicate. The S-HTTP draft specifies protocol extensions to inform the client that references to a specific URL (or responses to a fill-in form) are to be handled with security protocols, obviating the need to pass any potentially sensitive information without appropriate protection.

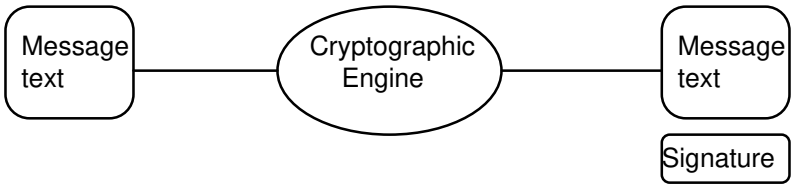
---

[67]See http://www.commerce.net/information/standards/drafts/shttp.txt for this draft. This is an Internet Draft, a working document of the Internet Engineering Task Force (IETF), and is subject to change. At the time of this writing, the December 1994 copy of the draft is the most current available.
[68]See *figure 3.2.1* for an explanation of this model.

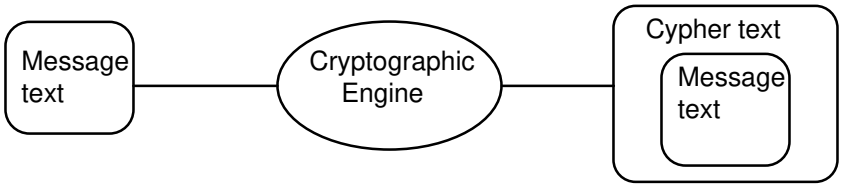
As with public key security, the model proposed in the S-HTTP draft allows the separation of signature and encryption. A message may be signed, encrypted, signed and encrypted, or not encoded. For example, consider the following diagrams:

Figure 9.3.2:

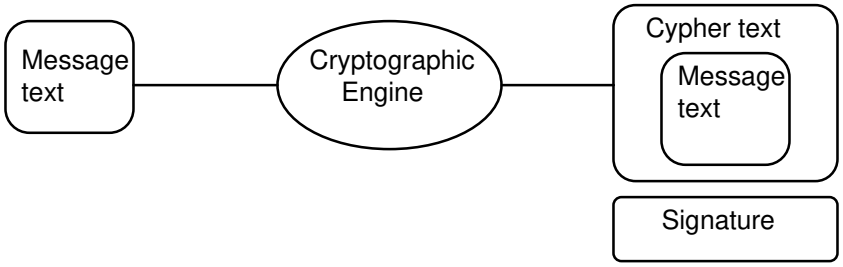
A signature is attached to text, using the sender's private key and the text of the message as input. This message can then be validated as originating with the sender by validating the signature with the sender's public key. However, it does **not** ensure that the message has not been read.



A message is encrypted, but not signed. This ensures that a message has not been viewed by other parties before it reaches the receiver, but it does **not** ensure that the message comes from any particular party.



A message is encrypted and signed. This means that it not only is the message available only to the receiver, but the sender is verifiable (at least to the level of trust in the key certification system).



A message may not be encrypted or signed. This is like traditional, pre-S-HTTP communications, where the message text is transferred in cleartext between the browser and server

S-HTTP is an ambitious undertaking. The attempt of the consortium is to provide a framework that can exploit the creative energy of the client and server writers while maintaining enough detail to permit interoperability among diverse implementations and cryptographic standards.

An alternate security model has been proposed by CERN and the European Union, and is called *Shen*.[69] The Shen proposal acknowledges that there are different needs on the Web and different levels of solutions to these needs. They break down the requirements into broad categories as follows:

- Weak authentication
- Strong authentication via public key
- Strong encryption of message content

This model further breaks down data into categories as:

- Public
- Copyright
- Confidential
- Secret
- Most secret

The Shen overview also provides a summary of the responsibilities of a browser and server with HTTP actions GET, PUT, and POST.[70] It is useful in describing needs of authentication and security in the World Wide Web model, but this proposal has been implemented on only one architecture under one operating system (Sun). Although initiated at the home of the Web, CERN, this proposal does not appear to have as much support as the S-HTTP initiative, nor does it appear to have captured the interest of client or server developers.

A third methodology has been proposed as a standard by Netscape Communication Corporation (NCC). This security model is called Secure Sockets Layer (SSL).[71] Unlike S-HTTP, SSL is not a modification of the HTTP protocol. SSL is, in fact, a much more general approach that provides security for protocols such as HTTP, FTP, and TELNET by interposing itself between TCP and the higher level protocols. Since security is not an issue that is unique to HTTP implementations, this is an attractive philosophy. The SSL protocol has been submitted to a World Wide Web security working group. It is important to track emerging developments with this protocol due to the meteoric rise in popularity of the *Netscape* browser and its corresponding influence on the Web community. Currently, this protocol is

---

[69]http://info.cern.ch/hypertext/WWW/Shen/ref/shen.html
[70]http://info.cern.ch/hypertext/WWW/Shen/ref/requirements.html
[71]http://home.mcom.com/info/SSL.html

implemented only in NCC's *Netsite* server and *Netscape* browser. *Netsite* is a commercial product, but the browser *Netscape* is free for some non-commercial uses.

Another alternative approach may be to develop applications that provide some degree of anonymity for the purchaser of goods or services. Here, credit cards might provide an audit trail that some users deem inappropriate. Designers of HTML applications may wish to have support for the concept of *digital cash*.[72] The idea of digital cash has been developed to provide a secure, reliable medium for conducting business through electronic channels while retaining the attributes of a physical currency transaction. Digital cash is one of the payment methods that is under investigation by OpenMarket, and other Web storefront service providers are likely to follow suit. Such commercial applications and the concept of the Internet storefront are discussed further in section 10.2.

Clearly there are many details of still left to work out as developers create secure HTML applications. For instance, if a transaction is to occur using credit card numbers, how should this take place? Should there be an explicit signature process where the user authorizes a transaction, or is simple encryption and subsequent transport of a valid number across the network sufficient to enable a transaction? Full digital commerce opens up new avenues for fraud that application design will need to address. Some of the issues raised are cultural, not technical, and will require a period of trust-building in these applications much as the banking industry has seen with the acceptance of ATM services. Today, most HTML applications that support on-line ordering of goods or services require out-of-band confirmation. One novel application that does not follow this pattern is the system developed by Spry, Inc. to allow on-line purchases of their *Air Mosaic* product.[73] A user may freely download a crippled version of the *Air Mosaic* browser that allows limited access to non-Spry resources. If the user is satisfied with the product, he or she may order the full copy by entering credit card information on-line. Since the client and server

---

[72]See http://digicash.support.nl/ for an explanation of the digital cash concept and implementations.
[73]See http://www.spry.com/ for information about this organization and this product line.

support basic encryption, this does provide secure transport of the credit card information. It is not difficult to think that credit cards may be issued with public key certificates available upon demand.

9.4 - Concurrency

The World Wide Web was designed primarily for one-way traffic, as a tool to navigate diverse servers and data bases. Its application development facilities have grown rapidly, but are handicapped by the need to fit into the existing model of servers and browsers. As discussed in section 9.2, communication between these partners typically takes the form of an atomic request and response[74], and some problems are thus raised that are not encountered in traditional transaction processing environments.

One such difficulty is that applications that want update access to database engines may implicitly require the HTTP server to keep state information at the server, especially if the application wishes to lock a field or record during the update process. Consider the HTTP server that creates a new instance of itself (as it does in UNIX) to handle each new client request. Since each transaction is a request/response pair, the server will spawn a task to handle the request. That server sub-task is the client to the database engine, and any resource utilization or locking information that the database engine collects or creates will be identified to this server sub-task, whose life span is only long enough to process the request from the client it is handling. What identifies a particular browser? How are successive requests from a particular browser collected into a logical unit of work? To do this with current design technology requires extensive state information at the HTTP server so that successive instances of the server can read this state information and identify a stream of work as belonging to a specific client. This state information must be kept in datasets or other non-volatile storage that can be accessed by these further server invocations. In effect, the HTTP server becomes a transaction manager for the database, a task for which it was not intended.

To effectively solve this problem, the answer may be to employ a transaction processor as an intermediary between the server and the database engine, as well as to provide *some* standard facility for local data storage in the browser.[75] Alternatively, if the database engine's own internal transaction processing system is sufficiently capable, that facility alone might enable a stateful browser to effectively

---

[74]See *figure 3.2.1* for an example.
[75]See section 9.2 for one possible method to handle state information in a browser.

interact with the transaction processing environment. For example, a browser might need to retain a transaction identification number or other such information to identify a particular stream of work, or to ensure that a record locked for update is rewritten by the same client (or eventually purged by time-out logic in the transaction processor). Implementations of data input systems that rely on stateful HTTP server applications (i.e. information stored by CGI scripts or otherwise as discussed in section 9.2) will not have access to the transaction management and proactive problem prevention facilities found in full-featured transaction processing systems. Ideally, HTML applications for transaction processing would use the HTTP server as a conduit into the transaction processing system, providing a return path for appropriate state information to be kept in the browser.

To see what deficiencies lie in the simplistic HTML model, contrast the paradigm described above with a mature, widely distributed transaction processing system such as IBM's Customer Interface Control System (CICS).[76] CICS is a stateful transaction processing system. Transactions and programs are defined to the system and execute under the auspices of the task dispatching module. There is a rich application programming interface (API) that allows a program to request resources such as memory, temporary files, permanent files, terminal services, and local or remote program execution. When a program invokes the API and relinquishes control of the CPU (briefly), CICS in its role as a transaction processing manager looks at the entire environment, searching for potential deadlocks, resource bottlenecks, priority transactions, or requests that can be routed to other systems. If a transaction requests input from a human-controlled device (a terminal or client workstation), the transaction may enter a suspended state while other work progresses. Much information is retained about each transaction both for identification purposes and to prevent some expensive operations (such as security authentication) from being performed unnecessarily. This environment presents constructs that a client may use to group a logical unit of work that must be completed or rolled back if not completed successfully. Since the HTTP daemon is not designed as a

[76]CICS has been available for over 25 years, and is found on many operating systems, ranging from mainframe systems such as MVS to workstation operating systems such as OS/2. It is also available on IBM's own UNIX™ platform, AIX.

transaction processing server, it must utilize the facilities of such a server to provide for construction of robust transaction processing application systems.

The idea of *groupware* is popular in the software development industry today. Conceptually, groupware is a suite of cooperative applications that manipulate a common database. This can take many forms, from the complexity of such well-known commercial products as Lotus *Notes*™, to basic electronic mail applications and their extensions. The thought of building such application systems around the World Wide Web model is intriguing but is really subject to the same problems as the database transaction model discussed earlier in this section, and additionally this usage invokes the problem of spontaneous bi-directional communications. As it exists today, the World Wide Web standard supports browser-initiated communications. For full implementation of groupware with support of *concurrent editing* of data elements, it will be necessary to extend the protocol and client/server functionality to enable the HTTP server to decide (in response to any internal or external stimuli) that it needs to send a message using the HTTP protocol to one or more browsers. In practice, it is likely to be necessary to pre-establish the socket-based connection between the browser and server before this can take place.

Consider a system for group development of a financial budget and a shared spreadsheet. Assume that there exist browsers α, β, and χ, and a server σ. The server σ acts as the coordinating agent. If a user at Web broswer α makes an update, the user wants it to appear immediately in the appropriate fields on the displays of browsers β and χ. Assuming that the server σ employs the services of a transaction processor or database engine to permit updates to the budget data, the problem still exists of propagating the new information. Refer to figure 3.2.2 for a review of the flow of information in an HTML application. For an update to appear on β and χ, given the current model, their users would need to explicitly request an update from the server σ. Without such spontaneous updates, information on the screens of multiple browsers in a groupware application would contain invalid information until manually refreshed.

How can this problem be addressed? Recall from section 3.2 that servers now exist for personal workstations as well as UNIX systems. A browser in a concurrent application system will need to combine the communication functions of these servers and the user interface functions of existing browsers into a coherent package. This new browser must listen for updated information as if it were a server, and in turn either respond appropriately with information it owns or recognize that the display instance it has generated is dirty and must be dynamically refreshed. Effectively, this will require a fundamental change in the nature of both browsers and servers. Each will need to send information (as is done by current browsers) and receive information, performing data updates or responding with other hypertext information (as is done by current servers.). A World Wide Web populated by such new generation browsers and servers would see the distinction between browsers and servers disappear. This would change the conceptual model of the World Wide Web from its current master/slave relationship into a peer-to-peer environment.

Some experimental work on a related concept has been done at NCSA. They have proposed a specification called Common Client Interface (CCI) that defines a standard method for Web browsers to interact with other applications.[77] The intent of this specification is to formalize a communication protocol that can be used by other application programs (on the same or different machines) to extract information from the browser (or from the Web itself). Currently, this only available using TCP/IP with version 2.5 of the UNIX *Mosaic* browser. NCSA envisions that CCI may eventually also use DDE and OLE in the Microsoft Windows environment, and Apple Events on the Macintosh. The focus of this work, however, appears to be on building a better-integrated client environment to more closely interoperate with supporting applications such as word processors, spreadsheets, and database managers. At this time, CCI does not directly address the issue of spontaneous refreshes of the client by the server.

[77]http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/CCI/cci-spec.html

9.5 - Development Tools

There is a wide range of tools available today to assist a developer in creating HTML applications. There are macro *add-ins* for popular word processors[78] as well as full-fledged standalone editors. These tools are available for virtually all popular workstation operating systems. What differentiates these programs, however, is the level of support they offer in the use of HTML. At the simplest level, HTML can be created with any plain text editor. This strategy depends solely on the expertise of the author to conform to valid language constructs. However, most tools assist the programmer by providing commands (or buttons) to place HTML directives around key text elements, much like a word processor typically provides commands to change the attributes of selected elements of text by applying such characteristics as bold, italic, or underline. This ensures that begin/end tags are matched, and that the proper punctuation and spelling is applied for the directives. What is often not assured, however, is whether the directive may be used in the current location without creating problems for browser interpretation. Recall that the formal specification for HTML is a DTD that defines not only the syntax for individual directives, but actually defines what constructs may be used in a particular place in a document and defines what constructs may be embedded within others. It is this level of support for application authoring that is most helpful in creating HTML that can run on any of the Web browsers available.

Some HTML editors attempt to be What-You-See-Is-What-You-Get (WYSIWYG), but it is more common for the editor to simply embed the directives into the text and offer the facility to dynamically load a Web browser with the current local file as its input. In fact, it is often the stated design philosophy that editing tools allow the HTML developers to use their preferred browser for viewing of pages under construction. This allows the developer to choose the browser that best suits his development environment.

---

[78]This is a popular idea. There are several public domain and commercial macros for Microsoft Word, for example. Microsoft itself has promised such a product (and even an integrated Web browser) will be available in early 1995. Another such macro, also for Microsoft Word, is *SGML Tag Wizard*, which can be found at http://www.unige.ch/general/tagwiz/taghtm.html

However, this again places the developer at the mercy of the tolerance of his preferred browser for deviant HTML usage.

As explained in Section 9.1, a Web browser interprets the HTML it receives dynamically to create its displays. There is no compile stage in code development and therefore run-time syntax errors can easily occur during interpretation of HTML pages. One answer to this problem is to provide some language validation facilities in the editor itself. One such editor that attempts to do this is called HoTMetaL, and is distributed by SoftQuad. Like the quasi-commercial browsers, this comes in two version -- one "professional" version supported by the developers, and another freely distributed version that is unsupported. HoTMetaL, available for Microsoft Windows and for UNIX windowing environments, not only assists the programmer in creating HTML pages, but also verifies the language usage against the HTML DTD. Actually, this program is capable of using various DTD's. When it was first distributed, early users found that few pages that had been written manually would actually load into HoTMetaL under strict adherence to the HTML DTD. The SoftQuad soon released an alternate DTD for use with "old" HTML to permit more flexible language constructs. Another option is to use a syntax verification tool that is run against HTML created with a traditional text editor. If an HTML developer chooses not to use such a tool, he should endeavor to test his applications against as many different browsers as he has available. Just because a page will function in one environment is no guarantee that it will be usable universally.

Current trends in software engineering place a great deal of importance in object oriented constructs and other methods of providing reusable and robust code. This is an area that does not necessarily map cleanly onto the World Wide Web model. While it is certainly possible to use these ideas and the tools that support them in creation of CGI scripts, the nature of HTML pages currently makes these pages poor candidates for objects constructs. An HTML page is a stream of commands in ASCII text interpreted in a linear fashion. There have been some attempts to create "libraries" of standard graphical objects that can be accessed across the network, each referencing the original server, but the logistical

constraints make this impractical.[79] At a basic level, however, simple directives are interpreted by the client and must be present directly in the stream of HTML processed by the browser. One could imagine a scenario where all pages are created dynamically by a CGI script using an object language, but this still does not address the problem of HTML consistency, since the system would still have to police inappropriate combinations of these objects. Ultimately, the HTML language is read and rendered serially by its browser. More significant changes would likely require a redefination of the paradigm.

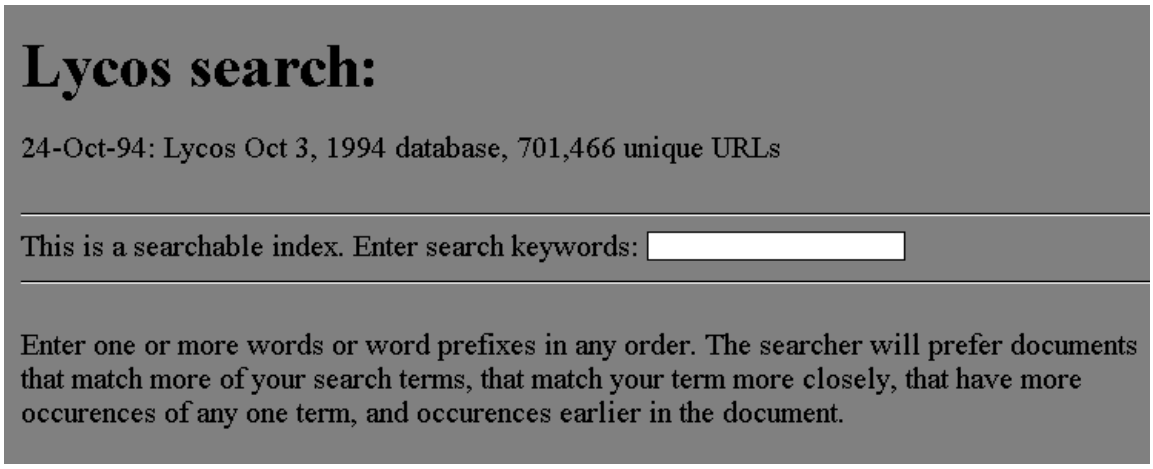
[79]Managing many simultaneous connections with adequate response for each client connection requires a powerful server and high network bandwidth.

Chapter 10 - Survey of Existing Applications

10.1 - Research

The World Wide Web places a great deal of information at the fingertips of a workstation user. Since the Web offers access to its own native information base plus that of Gopher and WAIS, a researcher can navigate many indices looking for relevant material on topics of interest. As briefly mentioned in section 3.1, there are numerous indices of the Web. Of central importance to this discussion is the fact that these tools combine interactive forms with database access. These databases are often quite large, with complex search engines[80] returning relevant records. Usually, these references in the database are rendered as hypertext links so that the researcher can then quickly navigate to the source. If it is a dead end, then the browser's history file will allow the researcher to quickly return to the list of matching records. Consider the *Lycos*™ search engine[81]. It has an index of several hundred thousand entries, and is rapidly growing. It looks like this:

Figure 10.1



Other indices are available. For a good example, the interested reader can consult the "directory service" page from Netscape Communication Corporation's *Netscape* browser:

---

[80]Such a search engine might be a WAIS server, for example, or might be a database indexing and retrieval designed specifically for the application.
[81]http://lycos.cs.cmu.edu/cgi-bin/pursuit

"http://home.mcom.com/home/internet-search.html". This provides pointers to many indices. Some indices only capture the title of a page, while others select arbitrary text from within a page. One problem is that on common search terms, there can literally be hundreds of "hits", and the researcher must winnow away the spurious information.

There is much information on the Web, but the researcher must remember that there are no special qualifications necessary to be a publisher on the World Wide Web. All it requires is a personal workstation[82] and a connection to the Internet. Information published using the Web may be reliable and accurate, but currently, the user must rely on the reputation of the source itself since there is no official body to referee or sanction particular material. Likewise, if a user builds reference applications, there is no guarantee of future accessibility. Machines may be moved, may be retired, or projects that supported a particular source may be terminated. In short, the quality of the source is highly variable and the source is subject to temporary or permanent service interruption. There is no canonical catalog of every URL that has ever been published. These are factors that must be considered in building research applications or in using existing applications to search the Web. This is a problem with many citations in this paper itself. The Web is the best source of information about itself, and consequently, references must be made to URL's which it is hoped will remain valid for a reasonable period. However, the author acknowledges that these sources are of uncertain long-term reliability.

[82]An MS-DOS PC running Microsoft Windows, for example, or an Apple Macintosh will do. A UNIX workstation is better, but is not required implement a server.

10.2 - Commercial Applications

What is providing the energy for much of the research and investigation into HTML application development that is currently underway? Significantly, there is a growing community of commercial users that see the Internet and especially the Web as a way to provide valuable services, whether through information dissemination or actual sales of goods. This is often delivered as a forms-based HTML application, in contrast to simple publication of hypertext multimedia information. If an organization wishes to realize a profit on its ventures, it must identify customers and exchange payment information, usually requiring a secure forms-based interface for transfer of data. The drive to maintain and improve a suite of HTML pages and applications for competitive reasons may well prove to provide a stimulus for development long after the initial flush of excitement of new technology has faded.

There is a growing market for the "virtual store" operating on the World Wide Web. Organizations such as Open Market[83] are striving to create a niche by selling expertise in setting up and running an on-line store. The services offered in such an effort include setup of a storefront, application development for order-entry, credit card processing (and online verification), and even product creation. In this context, product creation means that tools developed by Open Market can be used to help vendors bill customers for information on a per-volume or per-time basis. Additionally, Open Market provides services such as summary statements, a virtual "shopping cart", and fine-grained payment information for buyers. Open Market is a part of the Commerce Net consortium[84], which is responsible for the draft specification for S-HTTP, as discussed in section 9.3. Currently, spontaneous online order-entry is not available without use of non-standard browsers or clear-text transmission of payment information. OpenMarket customers currently set up an account, including payment vehicle such as corporate billing or credit card charges. However, Open Market is working with Spyglass[85] to ensure that Open Market products are compatible with Enhanced Mosaic (based on S-HTTP), due in late 1994 on some client platforms.

---

[83]http://www.openmarket.com/
[84]http://www.commerce.net/
[85]http://www.spyglass.com/

OpenMarket has instituted several security schemes to provide authentication. They offer a simple user ID and password, but this is optionally augmented with a challenge response system. They can prompt the user for information that is likely known only to the user, much like the way a bank might ask for your mother's maiden name as confirmation for a transaction. This information, however, is still subject to interception and re-use. For the highest level of user authentication, OpenMarket uses one-time handheld remote password generators. These credit card sized devices generate pseudo-random numbers on a time-based algorithm, enabling matching server software to know what code each card should generate at a given time. When the OpenMarket system prompts for a "passcode", the user enters the card's personal identification number (PIN) and the number currently displayed on the card's output window. This code is good only for the current transaction and even if intercepted could not be re-used to spoof authentication.

There are other competing marketplaces. Another one is NetMarket[86], also offering similar services, providing an Internet presence to vendors without requiring the vendor to posses expertise in the underlying technologies. NetMarket claims to be the first organization that offered encrypted order-entry processing over the World Wide Web (in August 1994), a claim that is difficult to refute. Their offering uses public key cryptography to exchange payment information. This public key approach does require pre-registration with NetMarket and use of a properly configured version of *Xmosaic* supporting Pretty Good Privacy (PGP) encryption.

Many other small marketplaces populate the Web. These usually offer a catalog of goods along with a phone or fax number for transmittal of payment information. However, an increasing number of such concerns are simply offering their users the option to enter payment information (credit card numbers) into un-encrypted forms. These organizations take the stance that if a customer is willing to enter the number,

[86]http://www.netmarket.com/

then they will use it. Many types of goods are available, ranging from computer equipment[87] to books[88] to specialty foods,[89] to name a few.

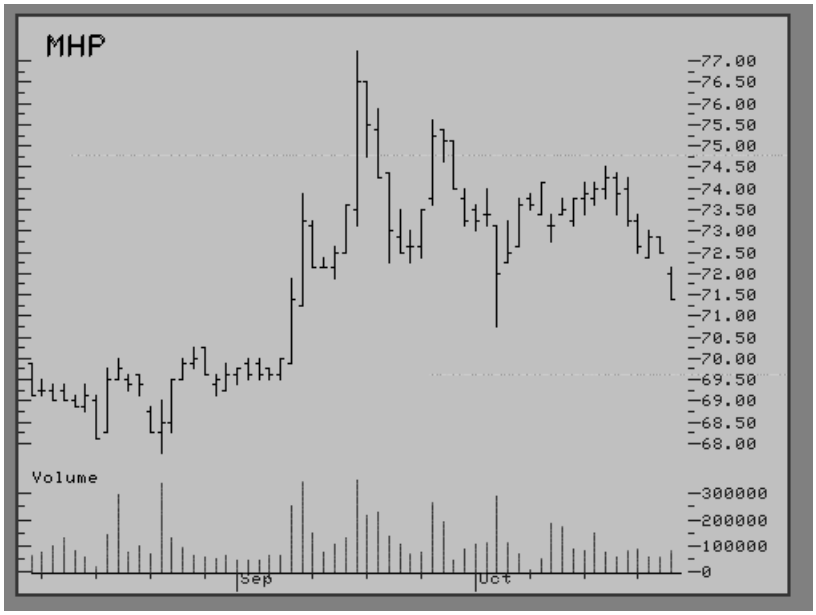
All these services share one common factor -- reliance on forms-based Web pages to allow the user to navigate through the system. In some systems without on-line payment systems, forms are typically employed only for searches of the catalog data. In others, however, the forms are used to collect demographic, payment, and product information.

Other commercial services are available. One common offering is the financial asset quotation facility. For example, some of these services can be obtained from the NETWorth[90] product from Galt Technologies. They offer fee-based inquiry into mutual fund prices (Net Asses Values) and offer graphs and other historical information. Others services are available from QuoteCom.[91] QuoteCom provides a free sample of their wares by allowing full access to information for one specific stock market ticker symbol, MHP (McGraw-Hill Publishing). Often, services that offer "free" information require that the user register and provide demographic information that can be used to create mass mailings or contact lists. The QuoteCom products can be sampled anonymously, and return both dynamic text and graphical data in response to entry of a ticker symbol. Shown on the next page is the output from their chart facility.

---

[87]http://cyberdeals.com/
[88]http://www.demon.co.uk/bookshop/
[89]http://www/infi/net/vadiner/vadiner.html
[90]http://networth.galt.com/
[91]http://www.quote.com/

Figure 10.2.1



This is an example of dynamically created graphical information. The user was prompted for the ticker symbol as well as other parameters pertinent to chart generation, such as the type of chart, colors, data to be plotted, and smoothing algorithm to be used.

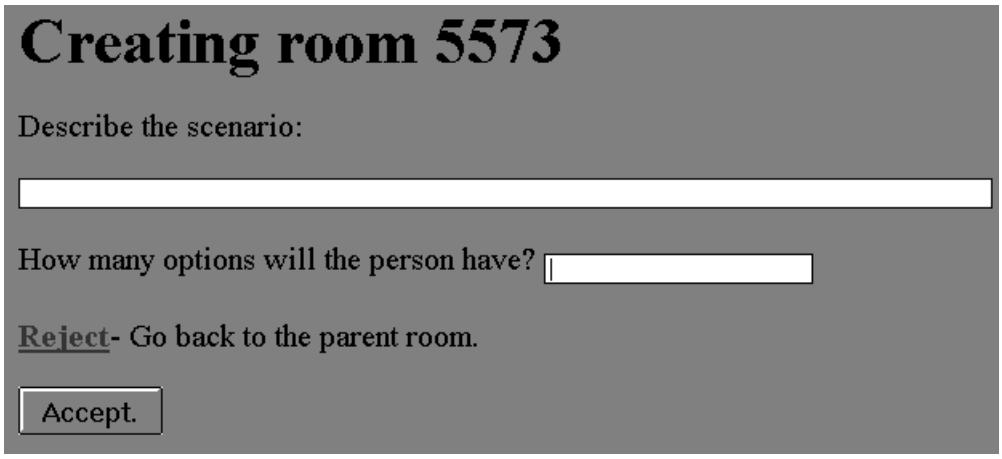
These are just a few examples of commercial services available at this writing. Others are appearing daily, and changes, especially in payment and authentication strategies, are occurring at a frequent rate.

10.3 - Games

There are numerous interesting games available on the Web. Since the nature of gaming requires input from the players, this is an ideal application for use of interactive forms-based HTML. The game *Addventure*[92] presents an intriguing twist on the "exploration" game. Here, the object of the game is not to search for a specific goal, but to seek out interesting threads in the game. When a player has followed a particular thread (plot) to its end, the player may optionally add a new "room" to the adventure. This means that the game continually evolves, and that it is diverse and often quite entertaining due to the large number of players that have contributed to the creation of the present game.

In *Addventure* each room is an HTML page. When a dead end is reached, a CGI script is driven that prompts the player to enter the scenario for the room, and the number of routes that will be available for subsequent players to use in leaving the room. This dialogue is as follows:

Figure 10.3.1



The input field for the scenario is open-ended, and a player can enter a quite lengthy description of the situation. Another form immediately follows this one, allowing the creator of the room to enter the description of the exit routes. The CGI script takes the input from these two forms and dynamically creates a new HTML page that describes the new room. The player can then enter the new room and continue to build on the thread or can leave that task for another player.

---

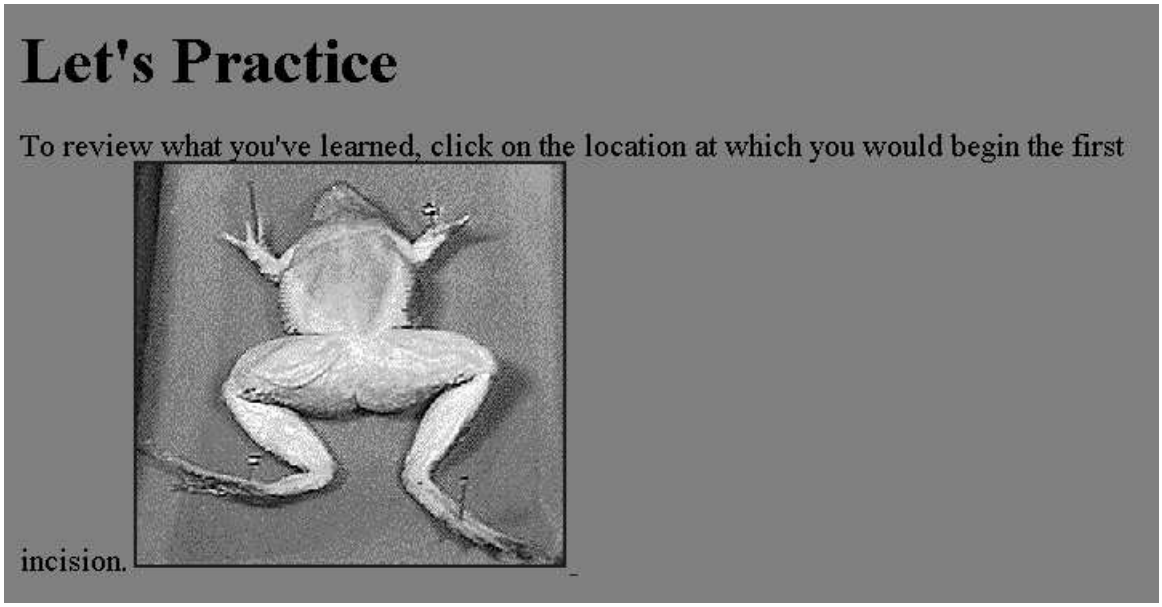
[92]http://acm.rpi.edu/~prisoner/addv-docs/

From the perspective of application development, the salient point is that this application continually grows and builds itself through the action of many players. A similar process could easily be used to build a knowledge base for troubleshooting help desk problems, for example. To be truly effective, such a model would need to allow for updates as well as page creation, but that could be accomplished off-line through manual updates to the created pages provided that the volume of erroneous data entered was relatively small.

10.4 - Computer-Based Training

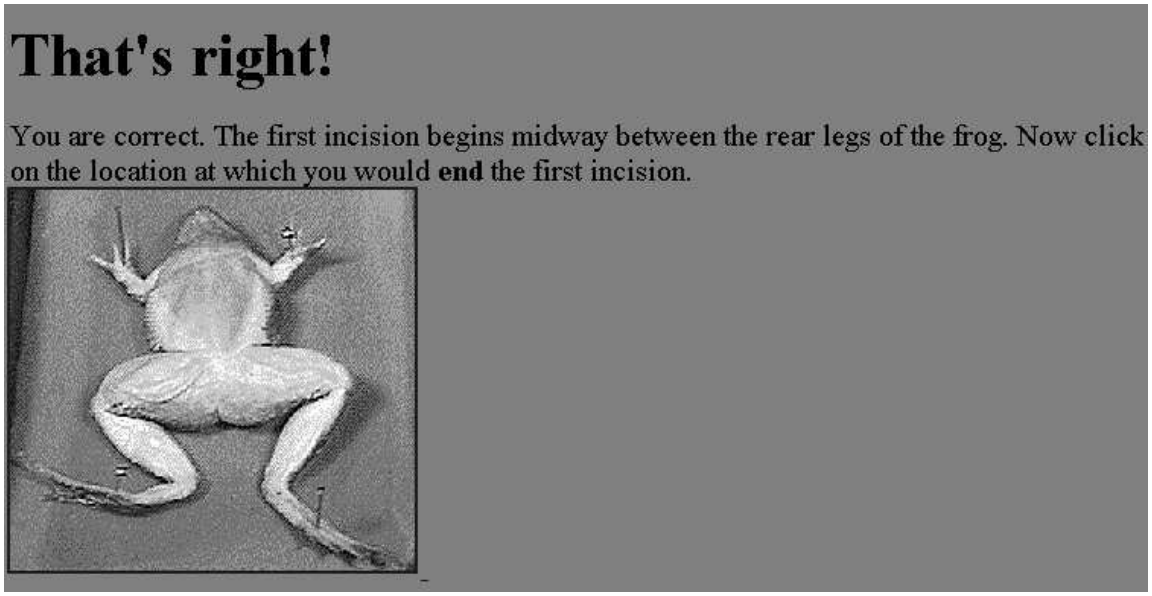
The Web offers many possibilities for education and training. Certainly, the multimedia hypertext presentation of information can enliven a subject, but to really provide feedback to the end user, the training developer must allow the user to enter information into the system. This could take the form of questions and answers....for example, check boxes could be used to deliver a multiple choice test to allow the user of a course to determine his preparedness on the material. This model can be extended in many ways, but one of the most intriguing applications is the "on-line" frog dissection tutorial at the University of Virginia, as briefly mentioned in section 6.1. Consider the figure below:

Figure 10.4.1



In this application, the user is first walked through the steps necessary to perform various parts of a frog dissection procedure. Figures 10.4.1 and 10.4.2 are from a set of diagrams dealing with skin incisions. In figure 10.4.1, the user is prompted to indicate the location of the first incision. Using *imagemap* techniques, the application senses if the mouse click was in the right area. If it is, the user is further prompted for a second incision, as in figure 10.4.2. If the user did not click in the right anatomical area of the form, a screen is displayed that asks the user to try again.

Figure 10.4.2



Here, the user is requested to select the location for the other end of the incision. A correct response delivers a page showing the next stage in the dissection. This is an instance where the state of the system is reflected by the URL accessed by the browser. If the server receives a request from a browser to access the imagemap supporting this diagram, that request implicitly carries the information indicating the user has correctly responded to the previous request. As we discussed in section 9.2, this is a technique that allows information about the state of the application system to be kept at the client instead of the server. Thus, the server can assume that if the user selects the correct location for the end of the incision, then the cut has been correctly identified.

This application provides an excellent example of in interactive computer-based training application that imaginatively uses the input capabilities of the Web. While it is easy at first glance to see the possibilities for training applications using forms input, an application such as this frog dissection tutorial can provide the genesis for other graphical tutorials. After viewing this application, it is possible to imagine a geography tutorial, for example. A medical tutorial would certainly be a natural outgrowth. This model sets a strong example for innovative development.

Chapter 11 - Conclusion

The World Wide Web provides an intriguing model for application development with the HTML language. It is currently possible to create many application systems as well as the simple hypertext information that provided the genesis for the Web. The inherent heterogeneity of the World Wide Web is a powerful attraction. As an application development environment, however, there are several potential difficulties that must be considered before creating mission-critical application systems using this concept. As we have seen, security is an obvious problem, but is one that is receiving considerable attention from developers of both client and server programs. Language standardization of HTML is a problem as well, but as formal standards are approved in this area, language difficulties should become less of an issue. Development tools are appearing to assist the programmers, but these are painfully immature compared to tools available for more traditional development environments. Severely limited locally accessible storage in the browsers currently forces data update systems to implement stateful server solutions, but this problem might be overcome by addition of richer storage directives to the language. The HTML developer today needs to be acutely aware of both the assets and limitations of the environment, and this paper has tried to draw these points to the attention of the reader. What may prove to be the biggest task, however, could be issues in the overall design of the Web client/server interaction. Chief among these concerns is likely to be the possible evolution of the master/slave interaction between client and server into a peer-to-peer relationship. This is a requirement if the users of the World Wide Web want to utilize this environment for development of complex, interactive applications that need concurrent access to information. This design change may not occur, and the users and developers may accept these limitations. However, the Web is a highly useful tool in its current form, and it may remain as it is, continually evolving incrementally to solve deficiencies and add to functionality to its protocol and programs.

REFERENCES

BURROWS, MICHAEL, ABADI, MARTIN, AND NEEDHAM, ROGER "A Logic of Authentication", Digital Equipment Corporation, *SRC Research Report 39*, Revised February 1990.

COMER, DOUGLAS, STEVENS, DAVID "Internetworking with TCP/IP, Volume III", Prentice-Hall, Inc., *ISBN 0-13-474222-2*, 1993, pp 107-108.

QUARTERMAN, J. S., SILBERSCHATZ, A., AND PETERSON, J. L. "4.2BSD and 4.3BSD as Examples of the UNIX System", *Computing Surveys*, Vol 17, No. 4: 379-418, December 1985.

SCHWARTZ, M. F., EMTAGE, A., KAHLE, B. AND NEUMAN, B. C. "A Comparison of Internet resource Discovery Techniques", *Computing Systems* **5**(4): 461-493; Fall 1992.

SMITH, S. W., AND TYGAR, J. D. "Security and Privacy for Partial Order Time", School of Computer Science, Carneige Mellon University, *CMU-CS-94-135*, April 1994.

VOYDOCK, V. L., AND KENT, S. T. "Security Mechanisms in High-Level Network Protocols", *Computing Surveys* **15**(2): 135-171; June 1983.