Mixed Retrieval and Virtual Documents on the World Wide Web

by

Christopher Alan Fuchs

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2, 1996

Author_____

Department of Electrical Engineering and Computer Science

May 2, 1996

Certified by_____

James S. Miller

Thesis Supervisor

Accepted by_____

F.R. Morgenthaler

Chairman, Department Committee on Graduate Theses

Mixed Retrieval and Virtual Documents on the World Wide Web

by

Christopher Alan Fuchs

Submitted to the

Department of Electrical Engineering and Computer Science

May 2, 1996

In Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

World Wide Web site managers are forced to serve data using *static retrieval* (from the file system) or *dynamic retrieval* (by executing programs) without the practical option of migrating between the two. It was found that dynamic retrieval through the CGI script mechanism has an overhead of 18-33% above its statically-retrieved counterpart. A *mixed retrieval* system allows Web documents to migrate between static and dynamic retrieval to gain the benefits of both methods and increase the overall performance of the server. In this thesis I explain why existing systems cannot support mixed retrieval; describe the components of a mixed retrieval system; propose a design for an initial implementation of a mixed retrieval Web server; present performance data verifying that mixed retrieval does not add significant overhead to current servers; and discuss the issues regarding future designs of mixed retrieval systems.

Thesis Supervisor: James S. Miller

Title: Research Associate, World Wide Web Consortium

Acknowledgments

There are many people who deserve thanks for helping me make it through this thesis.

Mark Day has been a tremendous inspiration. He continually pushed me to challenge myself, and reassured me when I doubted my work. I value the freedom he gave me to develop the project independently, while also guiding me when I took the wrong direction.

Jim Miller has also been a great source of encouragement for me. I especially appreciate the way he always helped me separate the important part of my work from the details. I thank him for taking me on as his advisee at the last minute, making time to always give me critical feedback, and giving me the chance to present my work to the World Wide Web Consortium.

The Workgroup Technologies group at Lotus Development Corp. played a pivotal part in making this thesis possible. Their support and open research environment made this project feasible. I especially am thankful to Irene Greif, John Patterson, and Dinko Zidaric for the ideas they contributed to this work, as well as to Candy Sidner for her support as the Lotus VI-A advisor.

All my friends deserve more thanks than I can express here, for putting up with me while I was working on my thesis. Finally, I thank my family for helping me get where I am today.

Contents

List of Figures

1. Introduction

1.1 The World Wide Web and other Information Systems

The World Wide Web *(the Web)* is a networked information system that allows platform-independent distribution and browsing of data. The Web defines a set of standard protocols and conventions that it uses to represent and transmit this information. It adopts a client-server model in which Web clients *(browsers)* request information (*documents*) from Web servers. Each request contains a Universal Resource Locator (URL), a scheme that represents both the Internet site of the Web server and the requested document location on that site. The HyperText Transport Protocol (HTTP), a simple, stateless client-server protocol, is used to handle this communication between Web clients and servers. HTML, HyperText Markup Language, is an encoding format native to the Web for representing information; hence, most Web documents are written in HTML.

Many companies use systems other than the Web for their primary information needs. Groupware systems such as Lotus Notes, or relational databases such as Oracle7, often are better suited to the primary information needs of the company. These systems are usually more structured than the Web, and allow a tighter control over the information itself [1]. Often, however, these information providers want to integrate the two information systems to gain the benefits of both: by using their primary information system for data generation, internal distribution, and access control, and using the Web for a more selective, global distribution, the two media are allowed to complement each other[2]. When it comes time to distribute that data over the Web, however, content providers encounter the challenge of successfully integrating the Web with their information system.

1.2 Data Retrieval on the Web

Two main methods of data retrieval are currently in use on the World Wide Web; these can be used to integrate the Web with another information system.

The first method of *static retrieval* simply retrieves documents from the Web server's file system, or a file system accessible to the Web server [3]. If legacy data is to be retrieved statically through the Web, it is necessary to publish this data to the Web server's file system. At the time of a Web request, this system is fast; however, it has other drawbacks. This method introduces the probability of stale data being served

over the Web. Additionally, depending on the amount of data being served, it requires a very large storage overhead on the Web server's file system.

Stale data and storage overhead can be overcome by using dynamic data retrieval. Web servers can serve dynamically retrieved (or *generated*) data through the Common Gateway Interface (CGI). CGI is a mechanism through which programs (*CGI scripts*) may be run by a Web server in response to a request, to generate or retrieve data from an external source[3]. URLs referring to a CGI script call are distinct from other URLs and may additionally contain parameters (such as in a query URL) to be passed to the script [4]; thus, a Web server can recognize that a request is a CGI script call and execute the corresponding script. The executed script generates or retrieves the data from some source unknown to the Web server itself, performs any necessary processing on that data, and sends it back through the server to the requesting Web client.

This method of *dynamic retrieval* has the advantage of allowing request-time retrieval of legacy data from its primary source. As a result, stale data cannot occur, and the server's file system storage requirements are low. Dynamic retrieval has the disadvantage, however, that executing a script is usually slow compared to simply retrieving the document from the file system [5]. This limitation cuts down on the number of requests a Web server can handle when dynamic generation is used, since it requires a large performance hit. As a result, it might not be possible to handle the required load of a fully dynamic integration.

Thus, both static retrieval and dynamic retrieval have different advantages when used to integrate the Web with another information system. A method should exist that combines the appealing features of both retrieval mechanisms while minimizing their disadvantages.

1.3 Mixed Retrieval

To achieve the benefits of both static and dynamic retrieval, the server can retrieve files statically in some instances and retrieve them dynamically in others. Specifically, the data being served should be partitioned into two subsets, one of which is being served statically and the other of which is being served dynamically. Correctly choosing the right subset for the documents allows the server to maximize the benefits of both retrieval methods. Therefore, methods for migrating between data subsets need to be explored.

Currently, the only way to migrate a document between these two subsets is to either change its request URL or manually redirect the request at the server. Changing URLs is impractical, however, since Web

clients should not be exposed to constant site changes simply because the data retrieval mechanism changes. Manual redirection is impractical because a large number of requests will probably need to be redirected, and the constantly changing conditions on the server will require a more real-time decision in order to correctly partition the documents into retrieval subsets so as to maximize the benefits of this method.

A *mixed retrieval* system preserves URLs to the client yet allows for an automatic redirection of certain requests to either their static or dynamic equivalents. The URLs in such a system then point to *virtual documents*, the retrieval method of which is unknown to the client. Therefore, such a system can reap the benefits of both static and dynamic retrieval, as well as allow seamless migration between the two.

Any number of mixed retrieval systems can be designed. The primary issue involves how the Web server decides when to retrieve documents statically and when to retrieve them dynamically. For this thesis, a cache model is applied to mixed retrieval to make this decision.

1.4 Mixed Retrieval Caching

In a *mixed retrieval caching* system, the Web server's file system acts as a dynamic document cache, storing the most frequently-accessed dynamic documents; all other documents are retrieved dynamically. A cache manager either publishes or removes data from the dynamic document cache depending on usage patterns, while maintaining a limited cache size within certain bounds.

Mixed retrieval caching therefore gains the benefits of both static retrieval and dynamic retrieval. Since some of the requests will be for statically-cached documents, it represents a faster system than one using solely dynamic retrieval. Further, since the size of the dynamic document cache is small compared to the size of all the data being served, the Web server's required storage overhead is smaller than that in the static model. Finally, the amount of stale data is reduced to only that data that is in the cache; because this subset is controlled, more stringent restrictions may be placed on its publishing behavior, allowing it to be published more often to eliminate all stale data; alternately, the cache management strategy can be designed so that frequently-modified data is not cached.

1.5 Design

The primary issues involved in designing a mixed retrieval Web server revolve around the process of identifying a request as a mixed retrieval request, then mapping that request to its static and dynamic

equivalents. Mapping a virtual document URL to its dynamic request not only involves translating the request URL, but also passing the correct information to the scripts through environment variables.

I discuss the tradeoffs involved in these problems and propose solutions. I explain an initial implementation of a Web server supporting mixed retrieval, and test its performance to compare it against a standard Web server as well as retrieval from a third-party information system.

2. The World Wide Web

The World Wide Web is a networked information system that allows platform-independent distribution and browsing of data. This chapter highlights the details of the Web's architecture that are relevant to the discussion in this thesis.

2.1 Data Representation

The data being sent over the Web can be in any format [6]; the Web itself simply defines a set of protocols for locating, storing, and transmitting that data, yet it does not place any restrictions on the *type* of data itself. Even though most Web pages are encoded in the HyperText Markup Language (HTML), data is not restricted to this format.

2.2 Client-Server Design (Flow of Control)

The Web adopts a client-server model to distribute data over the network [6]. This data is stored in repositories available to Web servers (most likely on the servers themselves) but not directly accessible to Web browsers. To see a particular piece of data on the Web, a user directs the browser to obtain that data by specifying its location. The browser locates or opens a connection to the server identified by that location and sends it a request for the data. This request is interpreted by the server; the corresponding data is located on the server, and is sent back (along with its data type) to the browser, through which it is displayed. The transfer of data is complete, and the connection may be closed. An abstract view of this process can be seen in Figure 1: Basic Web Client-Server Design.

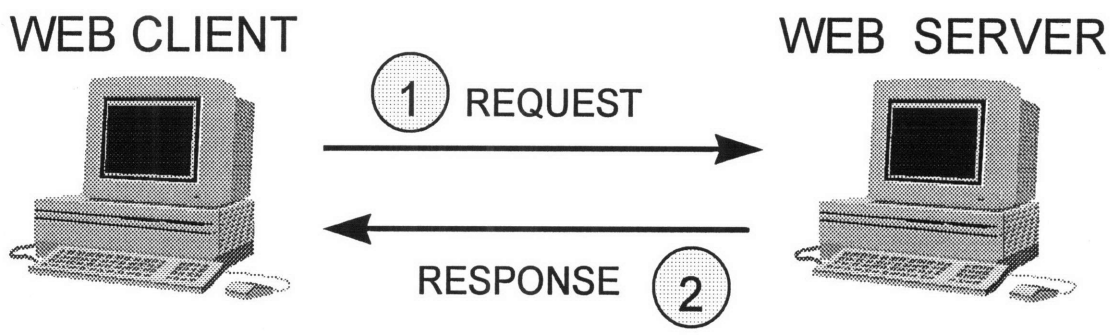

Figure 1: Basic Web Client-Server Design

2.3 Communication

When a Web browser sends a request to a Web server for a particular piece of data, it must be able to communicate with the server so that the server correctly understands the request. Additionally, the server must be able to correctly send back the data once it has been retrieved. The HyperText Transport Protocol (HTTP) is used as the most common method of communication between Web browsers and servers [7].

HTTP is a protocol that allows client-to-server requests and server-to-client responses. No communication occurs between servers or between clients. HTTP is stateless; a connection is opened at the time of request and closed once the data has been transferred.

2.4 Naming the Request

To make a request for a piece of data, a user must be able to name that data. Additionally, the Web server that stores that data must be located and identified.

The names of data on the World Wide Web are generically called Uniform Resource Identifiers (URIs), which serve as a general scheme for "naming, describing, and retrieving resources on the Internet." [8] Uniform Resource Locators (URLs) are a subclass of URIs that are used to address documents on the Web [9]; URLs will be the focus of this discussion, since other forms of URIs are either not fully developed or not in widespread use.

The syntax of a standard URL can be broken down into a number of components. In general, "a complete URL consists of a naming scheme specifier followed by a string whose format is a function of the naming scheme." [10] Therefore, in addition to naming the location of the document itself, URLs also name the

protocol to be used when accessing that resource. These can include standard Internet protocols such as FTP, gopher, and NNTP; the HTTP protocol, however, is most widely used on the Web to transfer data. [11]

A typical HTTP request URL takes the form [12]:

```
http://host [ :port]  [/path] [?query]
```

The "http:" section of the URL tells the browser that the request should be initiated using the HTTP protocol. The location of the Web server is given in the "//host [:port]" section, in which the server is identified either by its DNS (Domain Name Server) name or I.P. (Internet Protocol) address. Additionally, if the server is running on a port other than the default TCP port 80, it can be specified using the ":port" parameter. The "[/path]" part of a typical HTTP request URL specifies the location on the Web server itself (or a string that will map to that location) of the piece of data being requested. The final "[?query]" section is optional and allows a client to pass parameters to the server for dynamic retrieval requests (see Section 2.7, Dynamic Retrieval (CGI)).

For instance, the URL:

```
http://www.lotus.com/notes.html
```

would direct the browser to send an HTTP request to the Web server www.lotus.com; this request would contain the full path from the URL (/notes.html), which might direct the server to return the notes.html file.

2.5 Request Mapping

It is the server that is entirely responsible for mapping the [/path] section of a URL to a document or script. Upon receiving a request for data, a Web server parses the URL to allow it to internally locate or generate the file. The [/path] section of a URL gives the server the information it needs to retrieve the document. This section often takes a hierarchical form with the slash ("/") character representing a hierarchy delimiter. For instance, a company's Web site might be arranged hierarchically according to teams. In such a case, the Web page of the Marketing team might be accessed through a URL such as:

```
http://www.lotus.com/Company/Teams/Sales/International/homepage.html
```

This hierarchical arrangement is very convenient, since many file systems use the slash character to represent subdirectory structures. It is often assumed that URL mapping and directory navigation are synonymous because of this similarity; however, this is not the case.

This stems from the fact that Web servers map URLs to the exact location for retrieving the requested data. While most URLs approximate the location of the document on the server's file system, they are often augmented by additional path information leading to the specific files available to be served. For instance, a URL of the form:

```
http://www.lotus.com/internotes.html
```

might actually map to the file name /web_documents/internotes.html.

More complex URL mapping can also occur. Many Web servers support request *redirection*, in which a request is completely redirected to another request [13]. Servers may either send a response back to the browser indicating that a new URL should be loaded, or they may internally redirect the request and load the new URL. For instance, if a group of web pages were to be moved around a site, instead of changing all the links to those pages, it might be advantageous to simply redirect their old requests. Consider the case of Lotus Development Corp. upgrading their AmiPro word processor to the new WordPro word processor. Information on the Web about the older product might then be redirected to point to information about the latest release. Then, a request such as:

```
http://www.lotus.com/AmiPro/intro.html
```

might be automatically redirected by the Web server to information about their new product, with the request:

```
http://www.lotus.com/WordPro/intro.html
```

Additionally, any URL with a path matching the pattern "/AmiPro/" might be redirected to the corresponding location in the "/WordPro/" directory.

URL mapping, therefore, is used by Web servers to convert the path information in a URL to the actual internal location used by the server to retrieve the data. At any given time, a URL always maps to a single document's internal location and its corresponding retrieval method. URLs provide a many-to-one mapping of an abstract document name to the actual document data and retrieval method; multiple URLs may map to a single document through redirection. However, a single URL may never map to more than one document at the same time; the Web does not allow a one-to-many mapping of URLs to documents.

2.6 Static Retrieval

When an HTTP server receives a request for a document, most likely the location of the document is specified in the URL of the request [14]. The URL often gives the specific path to the requested document

on the Web server's file system; the server simply needs to translate the URL to a path (for instance, by appending it to the directory of servable Web documents) and retrieve that document from the file system. A basic overview of this process of *static retrieval* can be seen in Figure 2: Static Retrieval.

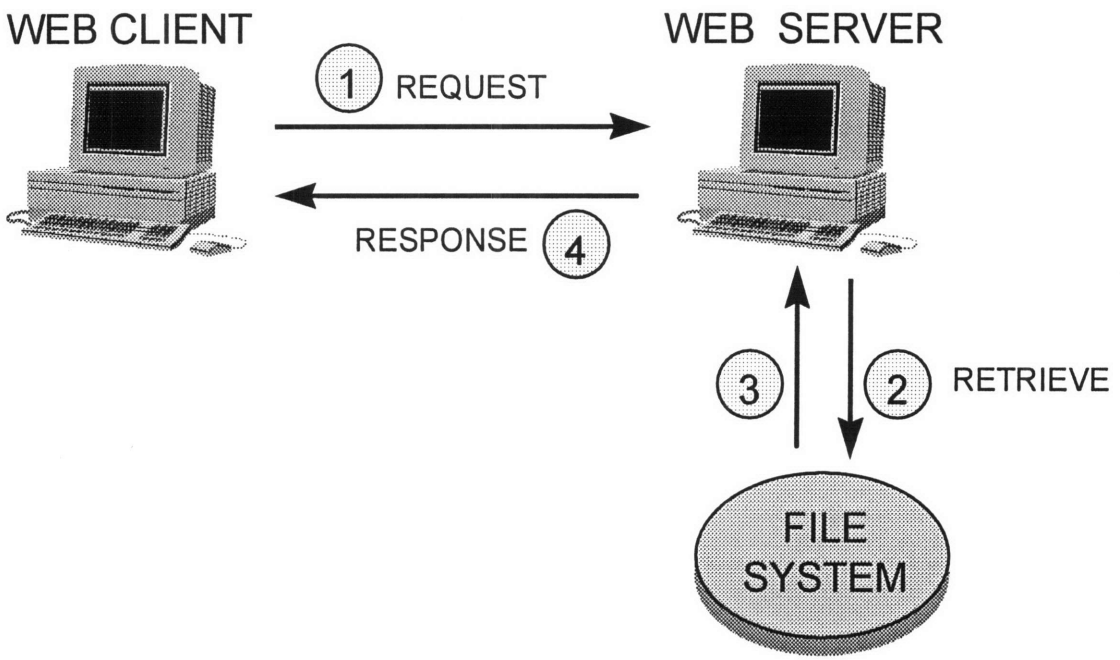

Figure 2: Static Retrieval

2.7 Dynamic Retrieval (CGI)

Most Web servers also serve dynamic data through the Common Gateway Interface (CGI). CGI is a mechanism through which programs (*CGI scripts*) may be run by a Web server in response to a request, to generate or retrieve data from an external source[15]. The executed script generates or retrieves the data from some source outside the Web server itself, performs any necessary processing on that data, and sends it back through the server to the requesting Web client.

A basic overview of this process of *dynamic retrieval* can be seen in Figure 3: Dynamic Retrieval.

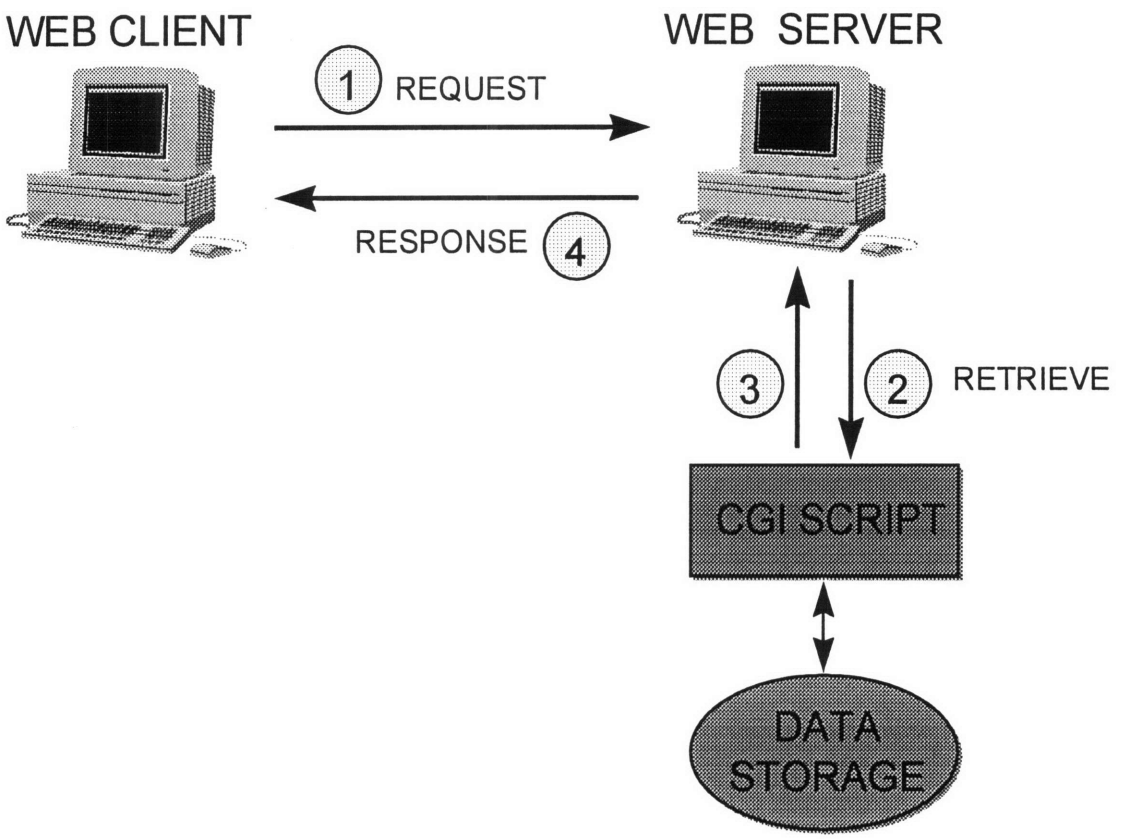

Figure 3: Dynamic Retrieval

2.8 Distinguishing Between Static and Dynamic Retrieval

A request for a dynamic document is delivered to the Web server as a simple HTTP request; however, the server should not simply return the CGI script itself to the client. A Web server must be able to distinguish CGI calls from simple static file retrieval. Additionally, security considerations require that CGI scripts be identifiable to the administrator; allowing arbitrary programs to run on a Web server is inherently insecure. Malicious users may be able to insert scripts hidden somewhere on the server's file system that could either destroy data or open security holes for remote users. Thus, it is necessary to be able to distinguish a CGI script from an ordinary file to be served.

Two main methods are used by Web servers after URL mapping to identify a request as a CGI script call: a distinguished directory or a distinguished suffix [16]. The first and most widely used requires that all CGI scripts reside in a special directory on the Web server. In this discussion, that directory will simply be

called the /cgi-bin directory. When a Web server receives a request for a URL contained in the /cgi-bin directory, therefore, it attempts to run the program rather than return the file. For instance, the URL:

```
http://www.lotus.com/cgi-bin/timepage
```

might instruct the Web server at www.lotus.com to run the *timepage* CGI script located in the /cgi-bin directory to generate a page containing the current time of day.

Through this requirement on CGI script locations, it is impossible for a malicious user to hide a CGI script from the administrator; only programs residing in the /cgi-bin directory are allowed to be executed by the server.

The second main method used to identify a request as a CGI script call is to require that CGI scripts end in a special suffix, such as *.cgi*. Therefore, any request for a file ending in the .cgi suffix will direct the Web server to execute the program rather than return the file. For instance, the URL:

```
http://www.lotus.com/timepage.cgi
```

might instruct the Web server to run the *timepage.cgi* script located in the root directory of the server's file system.

This approach has the advantage that CGI script location is not restricted to a specific directory on the file system; for a Web site that serves data from a number of users, this method facilitates individual creation of CGI scripts. Because of the requirement that all scripts end in a specific suffix, it also makes it possible for a site administrator to search through the site and find all of the CGI scripts that can be executed on the server. However, this approach is still less secure than using a distinguished directory access, since this directory can be controlled so that only the administrator may add files. With only a distinguished suffix, it is possible to sneak harmful CGI scripts into the server for a period of time until the next check is run.

Other features are common to both methods of identification. For example, CGI scripts may be passed parameters through the request URL to be used in their execution. In either approach, however, one thing remains constant: because the name of the program (as in the case of the suffix approach) and the location of the program (as in the /cgi-bin approach) are both reflected in the script request, URLs referring to CGI script calls are syntactically distinct from other URLs[17]. The job of identifying a retrieval method is therefore inherently bound to the request URL, and therefore visible to the client.

3. Integrating the Web with Other Information Systems

3.1 Motivations

Information systems serve to store data and provide some means to access that data; the World Wide Web is one example of an information system. Organizations may have numerous existing information systems with a long history of use and large amounts of legacy data stored within them.

The Web offers unique features that make it useful in certain situations; other information systems provide features that might prove useful in different situations. The main appealing feature of the Web is that it provides an easy method to globally distribute information. By simply setting up a Web site on the Internet, suddenly any data available from that Web server can be viewed from anywhere around the world. Its open client model allows anyone with a Web browser (which are available free and on virtually every platform) to view the data. It provides a means for enriching information with outside sources through its universal naming system of URLs by transparently allowing hyperlinks to sites other than the current server; similarly, it allows increased exposure and a somewhat collaborative environment for information retrieval, since anyone else with similar data can create hyperlinks to the data on their server.

To reap these benefits, one might choose to migrate completely from an existing information system to the Web. However, such a migration likely will prove impractical. In many cases, the Web model simply does not suit the type of data contained in the primary system. For instance, moving a relational database to a collection of Web pages does not make any sense; in addition to the overhead of converting all the data, such a migration would also mean losing the processing functions available through the relational database engine. Also, any number of unforeseen problems involved in data conversion or establishing a new architecture might become apparent. Instead, it is desirable to construct an *integration strategy* rather than a *migration strategy* to reap the benefits of the Web without losing the benefits of the original information system.

The two information systems will then most likely serve their intended purposes - the purposes for which they were designed and for which their features are appealing. The primary information system may be used for its current purposes without losing any functionality due to the integration with the Web; it can be the main repository for data generation and modification. The Web may be chosen to be used for a

more selective, global distribution; data generation and modification may be disallowed on the Web, depending on the integration strategy [18]. Still, the two media are allowed to complement each other.

When it comes time to distribute the legacy data from the primary information system over the Web, however, site administrators encounter the challenge of successfully integrating the Web with their information system.

While many factors need to be considered when determining how to integrate the Web with an existing information system, the context of this discussion will be limited to focusing on the data retrieval mechanisms available to Web servers. Specifically, it will address the problem of how legacy data can be retrieved through a Web front-end.

3.2 Methods for Legacy Data Retrieval

A number of techniques are available to a Web site administrator for dealing with the problem of retrieving legacy data from the primary information system. These techniques rely upon the different methods available to a Web server for retrieving data.

One technique is to use static retrieval; we can publish the information system's data to the Web server's file system, and retrieve these published copies statically. At the time of a Web request, this system is fast; a simple file system lookup and retrieval is all that is required. However, it has other drawbacks; depending on the amount of data being served, it requires a very large storage overhead on the Web server's file system, since all of those documents now have to reside there in order to be served statically. This might be countered by selectively publishing only some of the data in the information system; then, however, not all data will be available over the Web. Another disadvantage of using static retrieval is that no request-time processing can occur on the server side; this "publishing" method introduces the probability of the wrong data being served over the Web. If the data is published and then subsequently modified in the other information system, the data being served by the Web does not represent the same information as in its primary source. Such "stale" data can be combated by republishing; however, publishing represents a performance hit that might be better used by serving the data dynamically.

In some situations, these drawbacks are acceptable; if the site is not very large and if the data modification rate is small, this publishing method represents a viable solution with minimal drawbacks. For instance, a site that is used mainly as an information source about a company may not have very much data that changes often. A page of the company's stock quotes, for instance, might be the only page requiring republishing with any regularity. For large information systems with high modification rates,

however, the drawbacks to this publishing approach often render it impractical. For example, an information system that supports distributed collaboration would require most of its pages to be constantly updated if its Web users are to receive current data.

Another technique for retrieving legacy data is to simply create CGI scripts that know how to retrieve the data directly from the other information system. At each Web request, the CGI script would be run (with any parameters passed to it from the URL) to retrieve the data from the information system. This method has the advantages and disadvantages of a dynamic retrieval system; all of the data is current, since it is retrieved in real-time. Yet, this retrieval is slow, since it has the overhead of both a CGI script call as well as any necessary calls into the other information system. As a result, if all of the integration is handled by CGI scripts, the overall performance of the Web server is reduced, limiting the number of requests it can handle and somewhat defeating the purpose of putting the data on the Web by reducing the effective number of people with access to it.

The retrieval times of both static and dynamic documents were measured to determine an approximation of this effect. Static and dynamic versions of the same documents, of varying file sizes, were requested to determine the overhead associated with dynamic retrieval. These results can be seen in Table 1: Static vs. Dynamic Retrieval Times; the test conditions are described in Appendix A: Test Conditions.

Doc Size (bytes)	Static (s)	Dynamic (s)	Difference
0	0.08	0.10	0.02
2560	0.09	0.11	0.02
5120	0.09	0.11	0.02
10240	0.09	0.12	0.03
25600	0.11	0.13	0.02
51200	0.12	0.16	0.04

Table 1: Static vs. Dynamic Retrieval Times

These results show that using dynamic retrieval through the CGI mechanism limits the number of requests a Web server can handle within a given time interval, even when the script is not calling into another information system. CGI scripts running in the given test environment had an overhead ranging from 18%-33% of the equivalent static retrieval request. This overhead can be seen graphically as the retrieval time difference for a given document, in Figure 4: Static vs. Dynamic Retrieval Times. Therefore, for a given document using static retrieval, a Web server can support only 75-82% of these requests using dynamic retrieval. Even though different servers have different overheads associated with

them, this measurement illustrates the relative performance costs of static versus dynamic retrieval using CGI. This is especially illustrated since these CGI scripts were designed specifically to measure only the overhead of the CGI script mechanism, and thus represent a best-case scenario (see Appendix A: Test Conditions); scripts integrating into a real information system are most likely much less efficient.

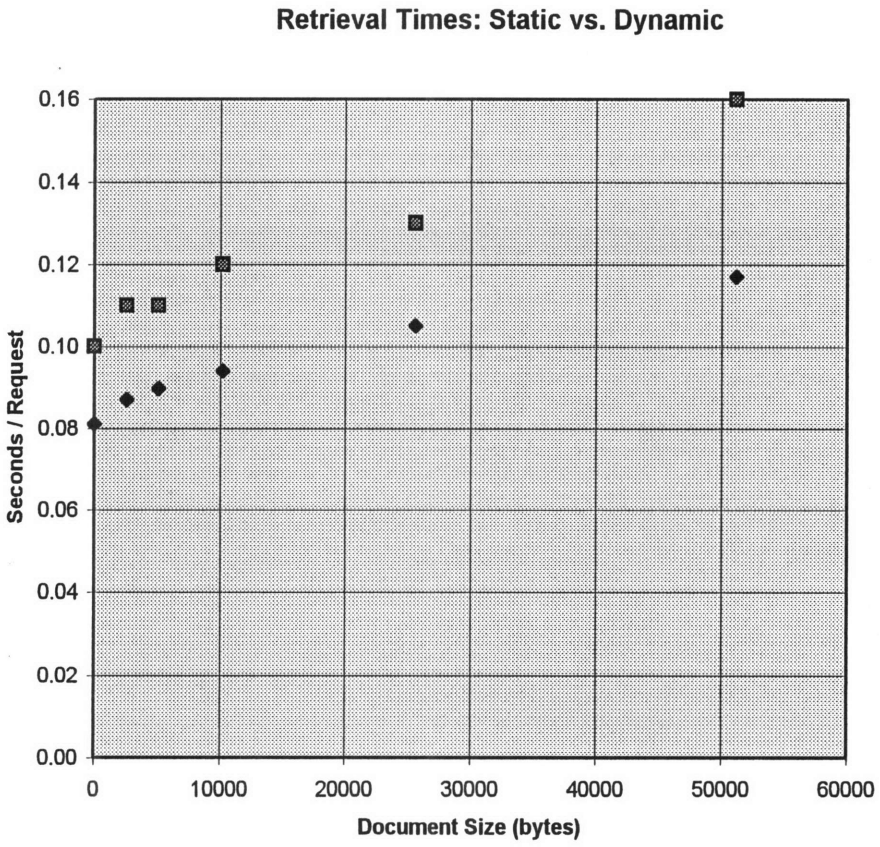

Figure 4: Static vs. Dynamic Retrieval Times

Static retrieval and dynamic retrieval both have their respective benefits to a Web site administrator. The best method of retrieval, however, is not always known in advance; in fact, it most likely will change over time to accommodate changing conditions on the server. A method should exist that combines the appealing features of both static retrieval and dynamic retrieval while minimizing their disadvantages.

4. Mixed Retrieval

4.1 Retrieval Method Should be Determined by the Server

To achieve the benefits of both static and dynamic retrieval, the server can retrieve files statically in some instances and retrieve them dynamically in others. Specifically, the data being served should be partitioned into two subsets, one of which is being served statically and the other of which is being served dynamically. Correctly choosing the right subset for the documents allows the server to maximize the benefits of both retrieval methods. Therefore, methods for migrating between data subsets need to be explored.

This partitioning should occur on the server side and should be isolated with an abstraction barrier from the rest of the system. Such a barrier gives the site administrator more control over the mechanics of the site and the contents of the documents, without having to worry about preserving links for the rest of the system (namely, the Web clients).* Changing retrieval methods, however, involves changing hypertext links and consequently breaking this abstraction barrier.

4.2 Requests Should Retain the Same URL

A Web site can be viewed as the system in Figure 5: The Web Site System.

* Uniform Resource Names (URNs), a concept along these lines, actually only isolates the retrieval mechanism from the client; it does not, however, establish an abstraction barrier that completely isolates the server. The retrieval mechanism is still bound to the URLs mapped from URN name servers.

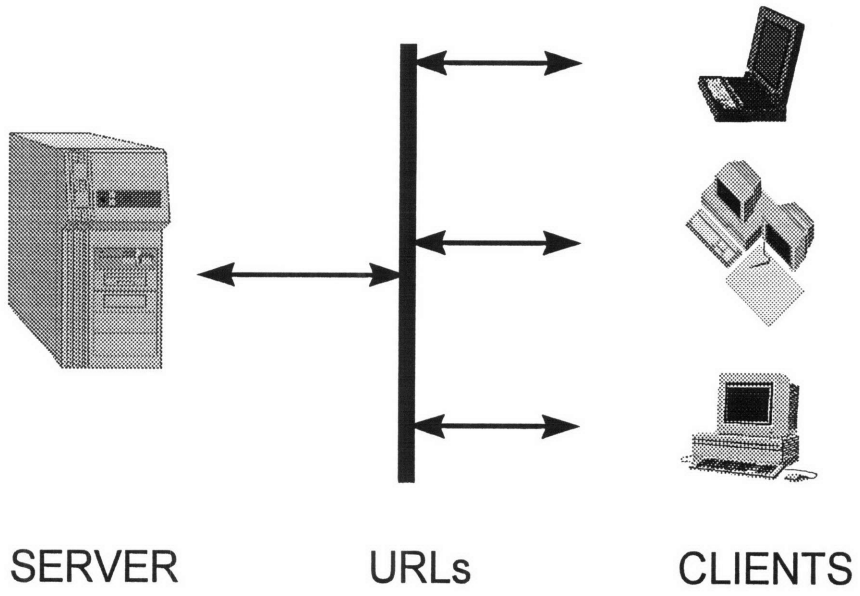

Figure 5: The Web Site System

The black vertical bar represents the interface between the server and the clients. This interface is defined by the hypertext links (addressed by URLs) into the site. As with any interface, its design should be specified and standardized so that both sides can work with it and the two subsystems can function together. Currently, the Web clients work with the interface abstractly; they assume that those links won't change, and don't have to take any special actions as long as the interface doesn't change. On the server side, though, this is not the case; restrictions on the URLs of the documents being served prevent the site administrator from leaving the interface alone if the retrieval method is to be changed.

4.3 Limitations Imposed by the URL Namespace

The current design of most Web servers imposes restrictions that prevent using either static retrieval or dynamic retrieval for a single URL request. Because of the security concerns associated with CGI scripts, all CGI scripts must either reside in the /cgi-bin directory or end in a .cgi extension [19]; therefore, requests that run scripts must have /cgi-bin as the top level hierarchy in the URL or end in .cgi. Conversely, any URL in the /cgi-bin directory or with the .cgi extension will direct the Web server to run a CGI script.

It is true that an incoming request can be redirected to a different, internal request that might use a different retrieval mechanism. However, this redirection itself is static; either the redirection is programmed to occur (in the server's configuration file, for example), or it is not programmed to occur.

In other words, an incoming request is still a static request or a dynamic request, but never can it represent both.

The retrieval method of a document, therefore, is inherently bound to its name, which in turn is bound to its request URL. These dependencies can be seen in Figure 6: Data Retrieval Dependencies.

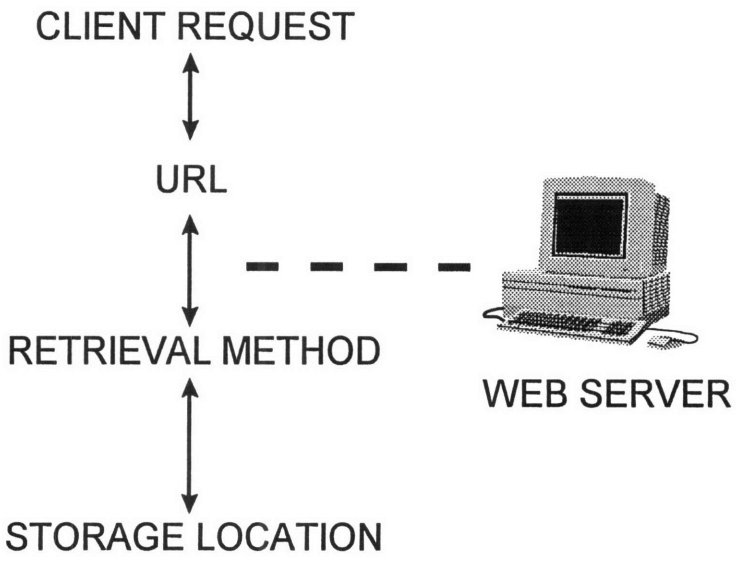

Figure 6: Data Retrieval Dependencies

Consequently, the space of all possible URLs that refer to static documents and the space of all possible URLs that refer to dynamic documents are mutually exclusive; a URL must always refer to *either* a static document or a dynamic document. This space can be viewed in Figure 7: Current URL Namespace.

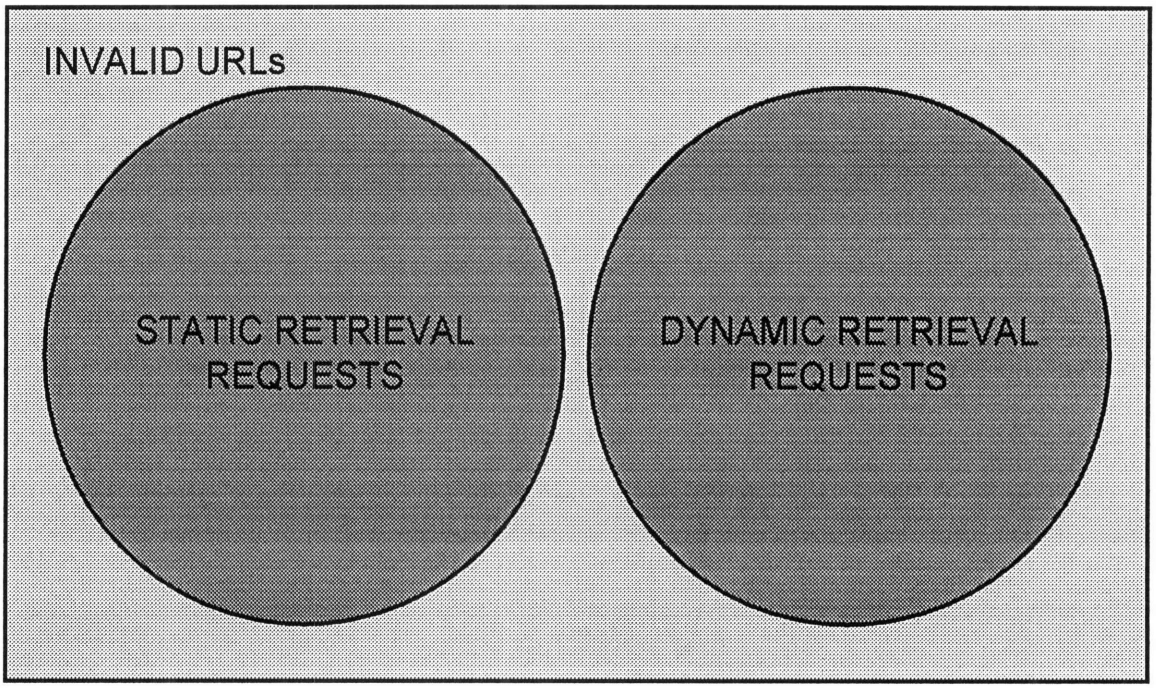

Figure 7: Current URL Namespace

Therefore, for the proposed goal, a single URL must be able to request to a document that is *either* statically retrieved or dynamically generated without sacrificing the security features of segregated CGI scripts. Such a namespace would then be characterized by the model in Figure 8: Desired URL Namespace.

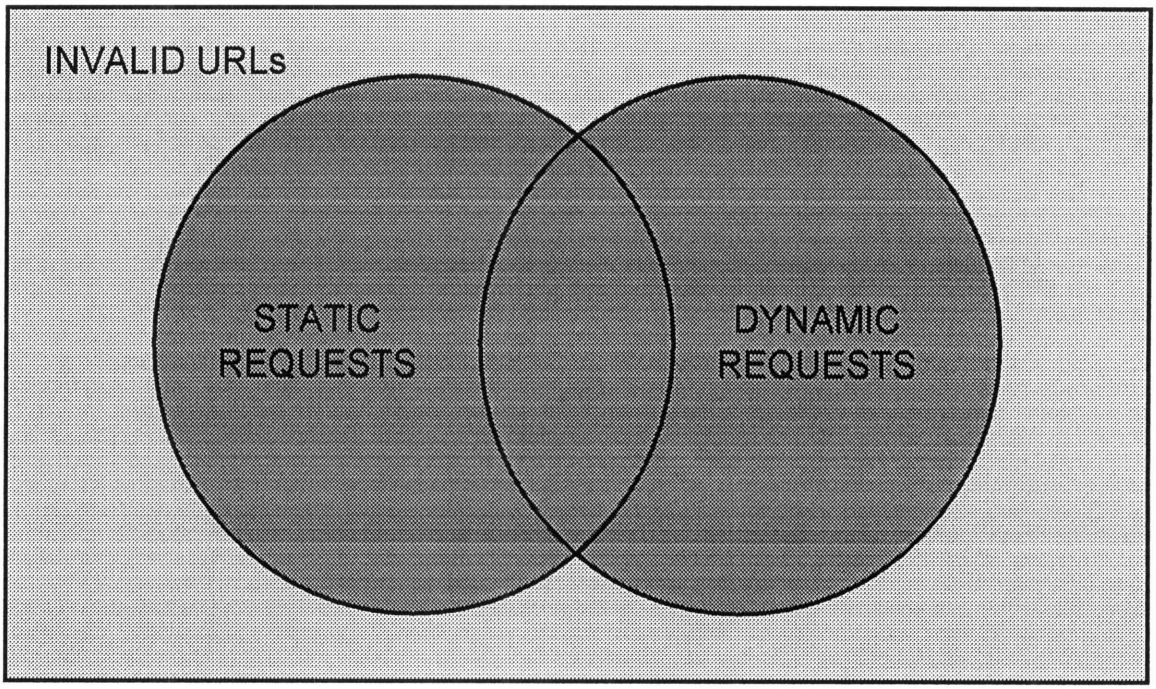

Figure 8: Desired URL Namespace

With this namespace, the retrieval method of a document is isolated from the interface - the names of the documents.

Such a namespace can be achieved currently by manually redirecting every incoming request to either a static retrieval request or a dynamic retrieval request. However, this procedure is impractical for two reasons: a large number of requests will probably need to be redirected, and the constantly changing conditions on the server will require a real-time decision to correctly partition the documents into retrieval subsets so as to maximize the benefits of this method.

A system is needed, therefore, that preserves URLs to the client yet allows for an automatic redirection of certain requests to either their static or dynamic equivalents. CGI scripts can reside in the /cgi-bin directory, yet they can be named by other URLs; the security of a centralized CGI script repository can be maintained, while the inherent binding between document names and retrieval methods is removed. Therefore, such a system can reap the benefits of both static and dynamic retrieval, as well as allow seamless migration between the two.

4.4 A Mixed Retrieval System

A system that meets these requirements can be termed a *mixed retrieval* system. Mixed retrieval allows client requests to stay the same by preserving URLs outside of the Web server, yet allows the server to automatically (internally) redirect incoming requests to either a static or dynamic URL. An abstract representation of a mixed retrieval system can be seen in Figure 9: Mixed Retrieval.

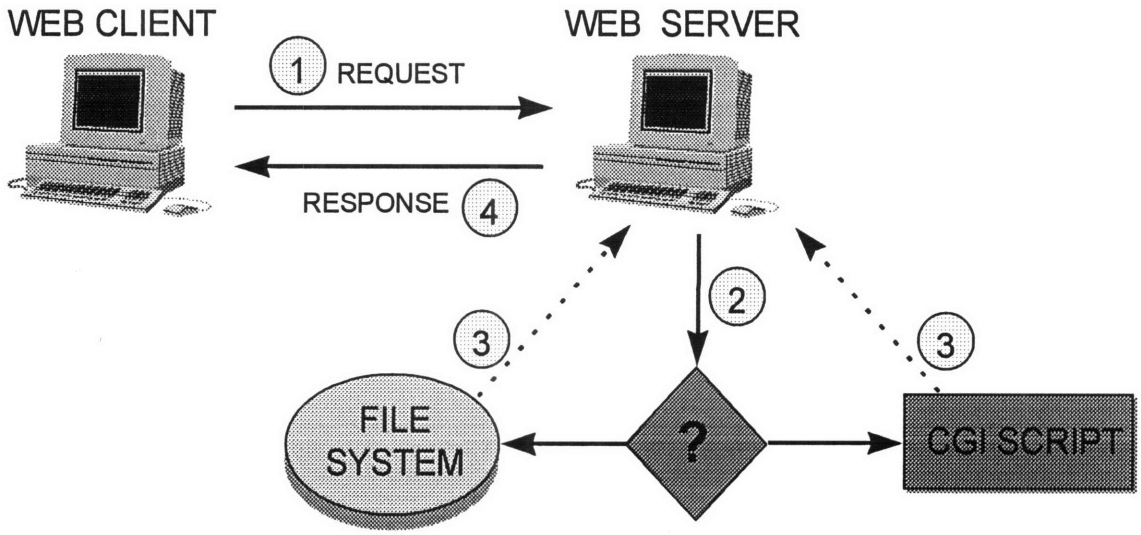

Figure 9: Mixed Retrieval

When a mixed retrieval Web server receives a request (1), it proceeds to make a run-time decision (2) to determine whether the document should be retrieved statically or dynamically.

Mixed retrieval therefore allows Web servers to become repositories of *virtual documents*, documents that are represented abstractly to the client through a URL that does not place limitations on the server's data retrieval mechanism.

4.5 Mixed Retrieval is not a URN System

Mixed retrieval is not an example of, nor a substitute for, a Uniform Resource Name (URN) system. While URNs are not currently in use, a number of proposals exist that detail their intended function. They should allow Web documents to be *named* by a browser rather than *located* [20], as is currently done with URLs (Uniform Resource *Locators*). Through some mechanism, the URN system should then provide a mapping from this name to a URL representing its location. It can perform this function by explicitly translating the name, or by transparently redirecting a URN request to its corresponding URL request.

URNs, therefore, eliminate the dependencies between a document's URL and the actual client request for that document; this process can be seen in Figure 10: URN Dependencies.

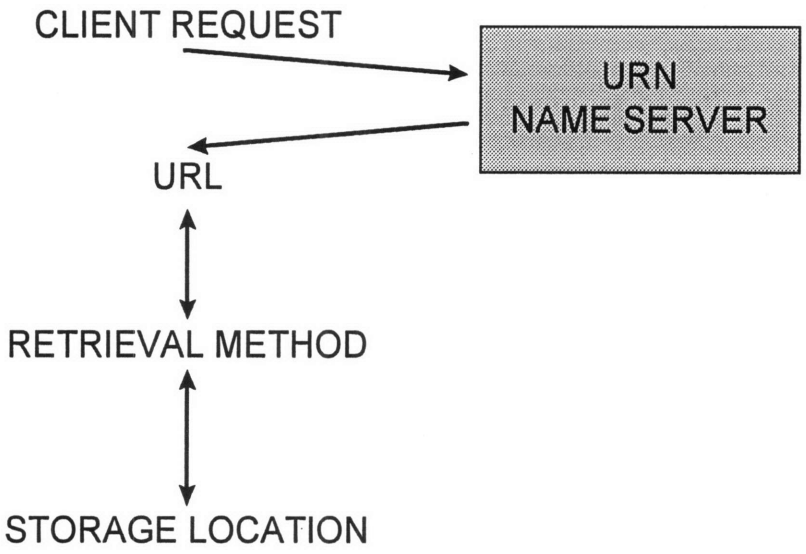

Figure 10: URN Dependencies

Mixed retrieval, on the other hand, differs from this functionality in that it maps one URL (representing a virtual document) to another URL (which implicitly contains its current retrieval method). This effect can be seen in Figure 11: Mixed Retrieval Dependencies.

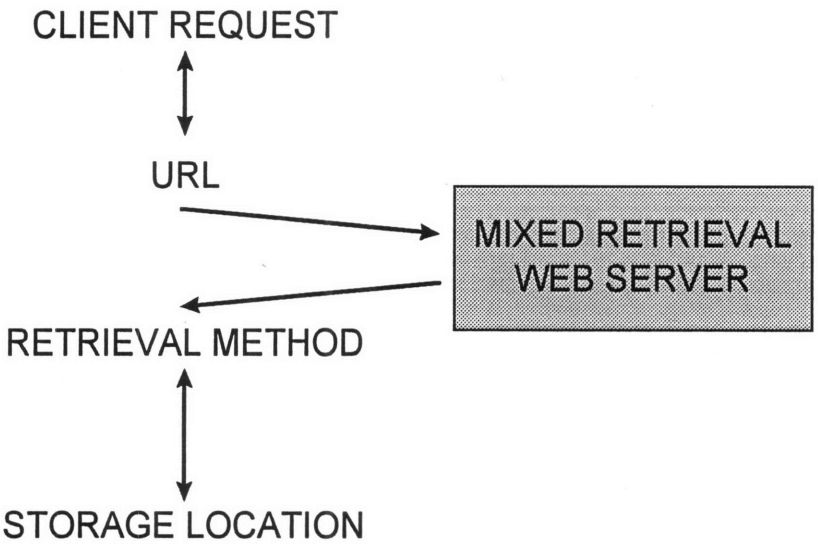

Figure 11: Mixed Retrieval Dependencies

In a mixed retrieval system, therefore, the browser's request must still contain the document's site, as well as the virtual path of the document; URNs would remove these responsibilities from the client request. Consequently, while mixed retrieval may augment a future URN system, it does not serve as a solution or replacement for it.

4.6 Proposed Uses for Mixed Retrieval

A number of possible situations, in which purely static or dynamic retrieval would not suffice, warrant the use of mixed retrieval:

- An existing information system contains diverse information that is to be distributed to the public. Most of this information doesn't change and can be published to a file system. However, some of it changes rapidly and must be accessed dynamically. It is unclear to the Web site developers what data will be modified frequently in the future, however, and therefore they want to keep open the option to either statically publish or dynamically retrieve any of the data on the site.
- A Web site administrator wants to migrate from a fully-published Web site integration with another information system to a dynamic integration method, without changing links or overall Web site structure.
- A very large information system needs to be made available through the Web; its entire contents cannot be published. It is necessary to publish as much as possible (based on what data is being

accessed the most), but still have available, through the same links, all the data in the information system.

Mixed retrieval can be implemented in any number of ways; the detail that needs to be resolved in each instance, therefore, is how the abstract decision-making process is realized: how does the Web server decide which data to retrieve statically and which data to retrieve dynamically?

5. A Mixed Retrieval Caching System

5.1 Overview

This chapter considers a specific instance of mixed retrieval, where the ability to choose static or dynamic retrieval enables the caching of documents. Mixed retrieval caching is analogous to virtual memory and instruction caching; in a mixed retrieval caching system, the Web server's file system acts as a dynamic document cache, storing the most frequently-accessed documents; all other documents are retrieved dynamically. A cache manager either publishes or removes data from the dynamic document cache depending on usage patterns, while maintaining a limited cache size.

5.2 Flow of Control

The flow of control of a mixed retrieval caching system can be summarized as follows:

- A Web browser opens a connection to a mixed retrieval Web server and makes a request for a document.
- The Web server checks the URL of the request; if it is identified as a virtual document request, it continues. Otherwise, it returns the document as it would if it were using normal static retrieval or dynamic retrieval, and the connection is closed.
- Once the request is identified as a virtual document request, the server parses the URL to a form that corresponds to a filename on its file system; this filename is used to look up the document. If the document exists on the file system, it is retrieved statically and returned through the server to the browser. The connection is closed. See Figure 12: Mixed Retrieval Cache HIT for an outline of this process.

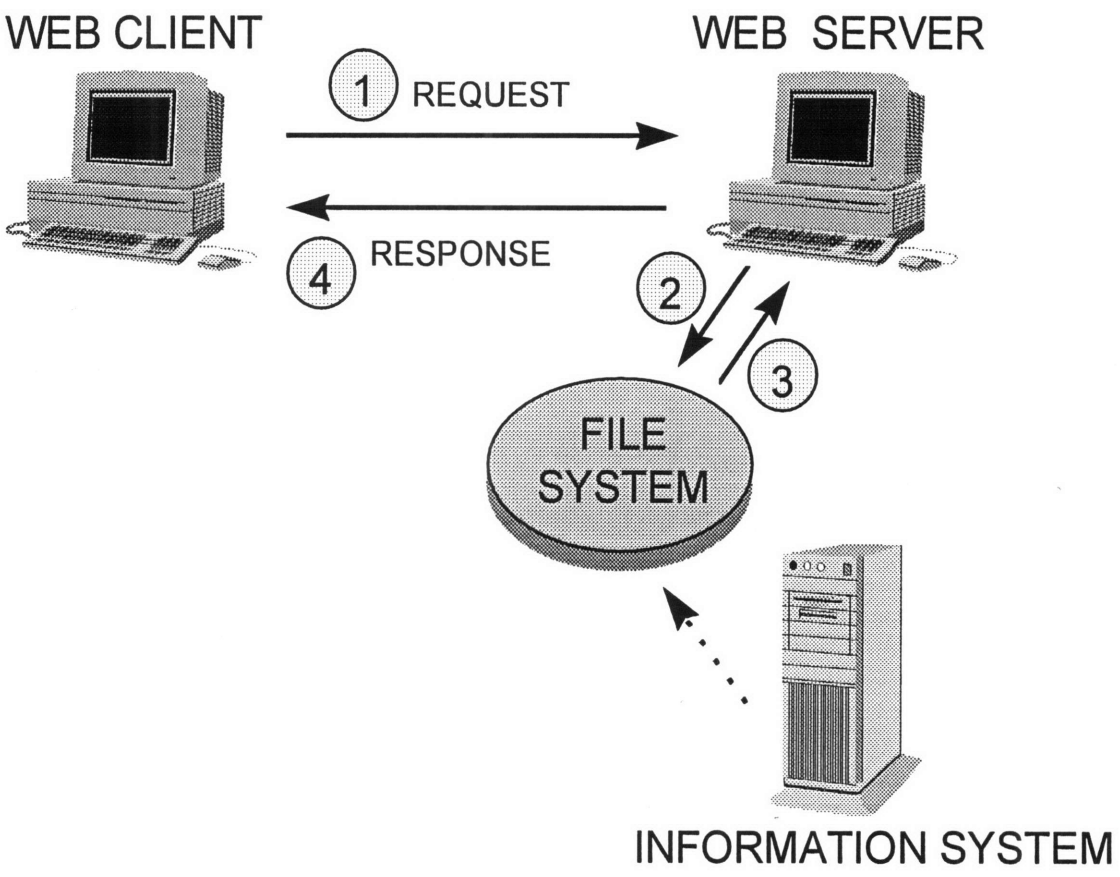

Figure 12: Mixed Retrieval Cache HIT

- If the document does not exist on the file system, the server parses the URL into a form that can be used to dynamically retrieve the document through a CGI script call. This script is executed.
- The executed script generates the results of the data as it would in a normal dynamic retrieval request. The server may optionally send a message to the cache manager, so that it can modify the contents of the cache.
- The server returns the generated document to the browser and the connection is closed. See Figure 13: Mixed Retrieval Cache MISS for an outline of this process.

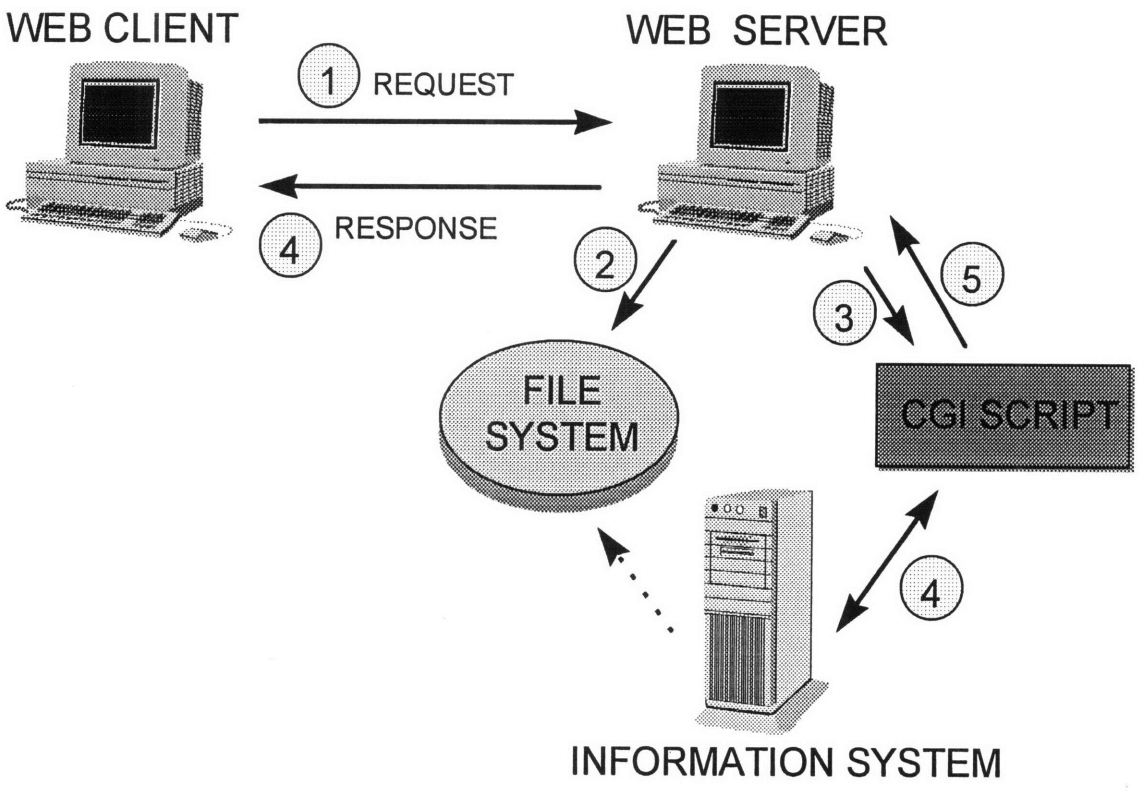

Figure 13: Mixed Retrieval Cache MISS

6. Design

6.1 Overview

Limitations in current Web server designs do not allow the implementation of mixed retrieval cacheing because of the inherent binding between document naming and retrieval. In this chapter, I describe a Web server extension that accommodates the necessary requirements.

To modify a Web server to support mixed retrieval caching, a number of design issues need to be considered. In general, these relate to the method used to partition the URL namespace. I first describe how to identify virtual document URLs, then I describe the methods used to map these URLs to either their static- or dynamic-retrieval equivalents.

6.2 Identifying Virtual Documents

A mixed retrieval Web server must be fully backward compatible with current URL specifications so it should be possible to replace an existing Web server without any modifications to the existing hypertext links or documents being served.

In this design, virtual documents are named the same way that static documents are currently named. It is assumed that the path in the URL leading to the document is valid and will result in a "virtual location" in which the document can either be retrieved statically or generated dynamically. This virtual location need not exist explicitly in the file system if it names a virtual path.

Since virtual document URLs look like static document URLs, there must be some server-side mechanism to determine which URLs actually represent virtual documents. This process involves partitioning the URL namespace into sections representing virtual document requests and normal static document requests. A major design issue, therefore, is the consideration of how to partition this space. A number of straightforward approaches will be discussed.

Our first attempt at a solution might be to bind each and every URL that should be a virtual document to an executable that can generate it. This approach has the advantage it is unlikely that any static document would be incorrectly treated as a virtual document; only through explicitly binding a document can it be considered virtual. However, this approach has a number of disadvantages that render it impractical.

Since one of the compelling reasons for using virtual documents is to help manage a Web interface to a large information system, it would be impractical to bind every possible document that might be accessed. Furthermore, the creation and deletion of documents from the information system would not be dynamically reflected in the Web server's bindings; it might miss data that should be virtual.

Since binding individual documents is too unweidly, we might bind all documents occurring within a given URL hierarchy. For instance, one could bind all documents with requests of the form:

```
http://www.site.edu/virtual/*
```

where * represents any document name. This approach has the advantage that it is much easier to bind a large number of documents to their dynamic generators. Additionally, the administrator does not need to know the exact URL of every document at binding time; only the location of those documents needs to be known. This scheme can even be slightly modified to allow wildcard hierarchies such as:

```
http://www.site.edu/virtual/*/*
```

This would allow even the hierarchy to be unspecified at bind time; only the URL pattern would need to be matched at the time of request in order for the Web server to recognize it as a virtual document. This URL-hierarchy approach has other disadvantages, however. Specifically, the documents within a given hierarchy may not all be of the same type; they might not be appropriately handled by the same generator. This drawback would not be a major deterrent if the goal is only to integrate the Web with a hierarchical information system such as Lotus Notes, in which the data (and hence data types) are organized into distinct levels of hierarchy. However, many Web sites use directories as a means for organizing heterogeneous collections of data about a particular topic; thus, a given level of hierarchy may contain heterogeneous elements. Even though it is possible for a generator to return more than one data type, this increases the complexity of the system. Therefore, while this might be an acceptable solution in some cases, in general it is impractical to assign virtual generators based on directory structure for most Web sites.

In our desired approach, some of the design requirements now become clearer. We will take the route of controlling the complexity of the system by allowing each generator to only produce one type of data. The virtual document identification process should have a larger granularity than binding individual documents, yet it should not rely on the document hierarchy either.

An approach that binds documents of a certain data type to generators for a certain data type meets these requirements. This solution has the drawback, however, that documents of bound data types might be considered virtual when they actually should not be. Even so, any real document of a bound data type that should not be considered virtual will never be considered virtual, since it will exist on the file system and

a generator will never be called. Additionally, any document that is not real (and hence does not exist on the file system) will result in a cache miss to the server. The server will call the generator for this document; if the generator is designed to interpret its parameters correctly, it will recognize this request as an invalid call and produce such a return result. Therefore, this potential hazard should never be realized.

In this design, virtual documents will be identified by their data types; the Multipurpose Internet Mail Extension (MIME) type of a document [21], which is already used by Web servers and browsers to establish file content types [6], will be used to represent these types. The server's administrator may register MIME types to be associated with virtual document generators.

If a client requests a document whose MIME type is registered with a virtual document generator, this executable is used to generate the document when it is not present on the file system. In the case where a document's MIME type does not have a registered generator and a dynamic document cache-miss occurs, the server recognizes this as a simple static retrieval request and returns the appropriate error for an invalid request.

In summary, the necessary modifications for this design are that the server must be modified to allow the binding of MIME content types to CGI-like executables, as well as to allow the server to check for these bindings at the time of request.

6.3 Mapping Virtual Document URLs to Static URLs

Since virtual documents are named in the same manner as static documents in this design, the URL of a virtual document may simply be used without parsing as the request URL of the dynamic document cache lookup.

6.4 Mapping Virtual Document URLs to Dynamic URLs

Mapping a seemingly-static request URL (as a virtual document is named in this design) to a dynamic request URL involves a number of considerations. Not only must the request be redirected to a dynamic request, but also the information present in the originally-called URL must be passed to the script to act as a parameter. It is often the case that CGI scripts use their request URLs to determine the exact piece or format of data to generate; if this information is not passed along, therefore, the scripts will not be able to operate transparently from a virtual request.

Two main pieces of information can be extracted from the original virtual document request. First, the original URL contains the request in the form specified by the browser. In normal dynamic retrieval, the QUERY_STRING environment variable is set by the Web server to this request [4]. This convention will be extended to virtual documents with the exception that the *original* URL be passed instead of the resulting dynamic request. The second piece of information that could potentially be necessary for the generator is the *translated* path of the resultant *static* URL request. For instance, if a generator needed access to data stored relative to the file system location of the virtual document, it would need to know that location. Similarly, if that document is ever to be cached, the generator (or a cache manager) needs to know the corresponding static location so that naming conflicts do not occur. Consequently, this parsed file system path is passed through the new QUERY_STRING_TRANSLATED environment variable to the generator. While this variable is not in the current CGI script specifications, it may be necessary for some scripts to allow for caching, since this variable contains the name of the virtual document if it were to be cached.

This design therefore impacts CGI script programmers in two ways. One requirement imposed by this design is that CGI scripts must be coded in order to accept not only parameters of the form of standard dynamic requests, but also of the form of a parsed virtual request. They must be able to interpret their calling URLs in both forms. While this places an additional requirement on CGI script programming, it does not pose any limitations on what those programs may or may not do. Secondly, passing the new information contained in the QUERY_STRING_TRANSLATED variable to the CGI script may be seen as a security risk; administrators must pay careful attention to those scripts employing this information for potential security holes or malicious activity.

6.5 Virtual Collections

It is necessary to subdivide the design to accommodate request URLs that do not end in a specific document request, but rather a trailing slash ("/"). Such requests are often either automatically mapped by the server to "welcome" documents representing the top page or index into that level of URL hierarchy, or are used to instruct the server to perform a directory listing of that location [22]. They represent a *collection* of links into that level of hierarchy. Based on this functionality and because of the difference in the requests, a mixed retrieval caching design should accommodate this through *virtual collections.*

In the mixed retrieval caching system, the slash ("/") character is still used as the URL hierarchy delimiter to represent a collection. While internal parsing might occur on the server side, the client names the

request for a collection simply as a URL with a trailing slash. This provides backward compatibility with existing naming practices.

6.6 Identifying Virtual Collections

Partitioning the URL namespace to bind virtual collections to generators is slightly more complex than binding virtual documents. In addition to the methods outlined for virtual documents, virtual collections can also be identified by their locations in the URL hierarchy as well as by their names. Still, this partitioning method tells us nothing about the types of collections that need to be generated. It is for this reason that this design takes the approach of associating a virtual collection with a specific data type. This data type is associated with a collection by associating a filename with each collection to be used when the collection is stored on the file system; this filename is of the MIME type of the collection, and represents the binding between the collection and its data type. With this data type then associated with the collection, its retrieval can be treated like a normal virtual document.

One additional problem involves actually determining how to name a collection as being of a certain data type. A collection is an abstract assemblage of items; consequently, its naming should be as abstract as possible. In this design, a collection is represented by a token-slash pair, such as:

```
Notes/
```

Token-slash pairs may also contain wildcards represented by the star ("*") character, for which any string may be substituted. This allows the classification of an arbitrary *level* of hierarchy as being of a certain type. This proves useful for applications in which the names of the directories may not be known but their locations in the hierarchy are.

Through this design for virtual collections, it is now possible to create *virtual directories* that may not even exist on the file system, but rather are interpreted on request by some collection generator. For instance, there would be no /Notes/ directory on the file system of the above example, yet because it would be registered to the server, a request containing the Notes/ level of hierarchy would appear to exist.

6.7 Mapping Virtual Collections to Static URLs

Requests for virtual collections are named in the same way as requests for static documents or directory listings ending in a trailing slash. However, since a virtual collection needs to be able to be mapped to a static filename for its cached version, internal parsing occurs on the server which actually maps a trailing

filename to the URL. Therefore, these requests are automatically redirected to the corresponding registered cache filename, which is retrieved statically if present on the file system.

6.8 Mapping Virtual Collections to Dynamic URLs

Since all requests for registered virtual collections are first mapped to their cached file name, this request is subsequently treated as a request for a simple virtual document. If a dynamic document cache-miss occurs, it gets mapped to a dynamic request in the same manner as a normal virtual document.

7. Implementation

7.1 Overview

An HTTP server was built that included support for mixed retrieval caching. This was done by using the CERN httpd 3.0 server [23] as a basis and adding modifications to the source code to support the new features. The CERN httpd was built by gcc version 2.6 with the World Wide Web Common Library of Code (libwww) version 2.17 [24]. Modifications were also made to this library's source code to support some of the necessary modifications. The server was built on an Intel Pentium-based processor under the Linux operating system running kernel 1.2.13. The computer was connected through a 10 megabit/second Ethernet 10-baseT connection to the Internet.

7.2 Configuration Architecture

The CERN httpd is configured through a text configuration file [25]. The server interprets rules in this file and behaves according to the constraints specified. Typical rules include commands for directing the server to add new MIME types or parse URLs. The modified server was built with the goal that an unmodified configuration file will result in an unmodified server. Only through new directives in the configuration file could mixed retrieval caching be activated. Specifically, new rules were defined to allow the administrator to register virtual documents and virtual collections based on the above outlined design.

7.3 Virtual Document Registration

Virtual documents are registered with the modified Web server through an AddGenerator directive in the server's configuration file. This rule has the syntax:

```
AddGenerator          input-mime-type     executable    output-mime-type
```

When the server receives a request, it first attempts to simply return that request from the file system. If the file is present on the file system, the file's MIME type is checked against all registered types of <input-mime-type>. The server returns the document from the file system, with the content type specified by <output-mime-type>. This allows generators to perform translation from one type to another and cache the translated type.

If the file is not found on the file system, the document's MIME type is checked against all registered types of <input-mime-type>. If the type is registered, the first rule with that registration is used as the virtual binding. The CGI script <executable> (which, by requirement, must be in the /cgi-bin directory) is then run to produce the document of type <output-mime-type>.

If the server receives a request for a document that is present on the file system but whose MIME type is not registered, it simply returns it using normal static retrieval. Similarly, if the request is not found on the file system and the MIME type is not registered, the server returns error code 404, *Not Found* [26].

7.4 Virtual Collection Registration

Similar to virtual documents, virtual collections must be bound to their corresponding generators. Registration takes the form of a new configuration rule of the form:

```
AddCollection          collection    cached-filename
```

The <collection> parameter may take two forms, to support wildcards, as illustrated in Table 2: Collection Registration Syntax.

AddCollection col/ index.typ	Registers col/ to be a virtual collection. Its cached filename gets mapped to col/index.typ. The collection's data type is that which is registered for index.typ.
AddCollection col/*/ index.typ	Registers col/*/ to be a virtual collection, where * can be any string. Its cached filename gets mapped to col/*/index.typ, where * can be any string. The collection's data type is that which is registered for index.typ

Table 2: Collection Registration Syntax

Internally, the AddCollection directive simply pattern-matches a request against <collection>; in the case of a match, it appends <cached-filename> to the request. The virtual document mechanism handles all further interactions.

7.5 Testing

The mixed retrieval server's functionality was tested in two main phases: unit testing and integration testing. The unit testing phase consisted of basic, controlled tests designed to verify that each new feature of the server worked as specified. The integration testing phase verified that the server performed as specified when run with uncontrolled mixed retrieval requests into a mocked-up database. The results of these tests can be seen in Appendix D: Test Applications.

8. Properties of the Dynamic Document Cache

8.1 Caching Through Mixed Retrieval

Mixed retrieval caching should be used with the goal of maximizing the number of Web requests that are in the cache while minimizing the size of the cache and the amount of stale data possible. However, because of its properties, mixed retrieval may be used for purposes other than a simple caching mechanism; in such a case, the cache manager may work with different goals.

When used as a caching mechanism, therefore, mixed retrieval gains the benefits of both static retrieval and dynamic retrieval. Since most of the requests will be for statically-cached documents, it represents a much faster system than one using solely dynamic retrieval. Further, since the size of the dynamic document cache is small compared to the size of all the data being served, the Web server's required storage overhead is much smaller than that in the static model. Finally, the amount of stale data possible is reduced to only that data that is in the cache; because this subset is controlled, more stringent restrictions may be placed on its publishing behavior, allowing it to be published more often to eliminate all stale data; alternately, the cache management strategy can be devised so that frequently-modified data is not cached.

8.2 Cache Management

When an information system has been integrated with the World Wide Web using mixed retrieval, it is necessary to devise a method for determining which data to add to the dynamic document cache, when to add that data, and which data to replace when the cache fills up. Such a method is called a *cache management strategy*. Many books have been written on strategies for managing a cache; this section only serves to highlight issues that are specific to the file system as a cache when using mixed retrieval over the Web.

8.3 Data Caching

A large number of factors may be used in determining the cache management strategy of a caching system. Often these factors are dependent on the type of data being cached and the data retrieval patterns of its users.

Frequency of Web access, frequency of data modification, and other data-specific characteristics need to be considered when determining which data to move to the dynamic document cache. Since the primary purpose of caching the data is to increase the overall performance of the Web site that is integrating with the information system, the selection of which data to move onto the cache should always bear this purpose in mind. While it is possible that some data might be better left on or off the cache, general trends can be observed that should, on the whole, increase the performance of the system.

Frequency of Web access to the data is one trend that should help dictate which data should be placed in the dynamic document cache. In general, in staying with the general cache model, data that is accessed frequently over the Web should be in the dynamic document cache. This approach assumes that if a piece of data has frequently been accessed from the Web in the past, it is most likely going to be accessed frequently from the Web at another time in the near future.

Spatial locality is another factor that can be exploited to help manage the cache. Given a classification scheme for the entire set of data available through the Web, it is possible to determine how "close" any two data points are in this space. A high number of accesses to a given subset of this data might indicate that there are likely to be more accesses in the future. By moving this entire subset to the dynamic document cache, the system might avert possible future performance hits. For example, if many people are requesting documents from a certain database from the integrated information system, it might make sense to move that database to the cache. While the server might not be able to intelligently determine how each document's content relates to the next, it can determine that they are all within that single database.

An added consideration when determining which data to cache derives from the fact that the information system being integrated with the Web is dynamic, and that its data might be constantly changing. This factors into the cache management strategy since it affects the amount of stale data that could possibly be in the cache. For instance, a piece of data with a high modification rate should most likely *not* be cached; since the cached version might not automatically change with the modification of its original copy, keeping this data in the cache could result in a higher probability of stale data being viewed over the Web. Additionally, if modifications were *automatically* propagated to the dynamic document cache, this might

present a performance hit to the Web server, since the cache might have to be updated very frequently. While data modification should be factored into the cache management strategy in this way, the general trend can always be overruled by special cases in which Web access to that data is extremely high, for instance.

8.4 Cache Size

Determining the optimal cache size is a process that is specific to the type of data being served and the resources available to the Web server. In general, a larger cache can speed up the system, yet it can result in the tradeoffs illustrated above.

When cache size is an issue, it is necessary to only keep in the cache a subset of the total data available over the Web. The cache should take advantage of the resources available on the Web server - namely, the amount of storage space available; however, the cache should not be so large as to defeat the purposes of a cache. For instance, if it were the case that a larger cache required a longer lookup time, this factor would need to be balanced against the benefits of a larger cache. However, since this is not the case in most file systems, it should not factor in determining the cache size. It is necessary to identify and quantify such tradeoffs, however, before determining how large to make the cache.

In many cases, the cache size is not the limiting factor in determining how to serve the data. Systems might employ mixed retrieval simply for avoiding stale data in "hot" locations - those subsets of data that are frequently modified. Their web site may well be able to adequately handle the contents of its data even if the entire set of data were statically published. However, because a finite subset of that data might be frequently modified, it might be desirable to keep that data out of the cache. In such a system, therefore, cache size is not an issue.

8.5 Data Replacement

In all systems, the limitation of available resources poses a problem when the bounds of those resources are encountered; in the case of a dynamic document cache, it is limited by size, whether because of the physical storage capacity of the device it is on, or because of scalability issues that prevent it from getting too large. As in all cache systems, an important issue of replacement involves *which* data to remove from the cache when new data must be inserted. This is discussed at length in various texts on caching and will not be covered here. In the case of a dynamic document cache for a mixed retrieval Web server, however, there are other specific issues that should be considered, in addition to those that are relevant to general caching systems.

Ejecting data from the cache on a mixed retrieval Web server involves the actual deletion of data from the file system, since it needs to free up space for inserting new data. This could potentially present a security hazard if the cache manager has the right to arbitrarily delete data from the Web server. A number of solutions to this problem exist. The most secure solution is to not allow the cache manager to delete any data; this allows the cache manager to *overwrite* files that are already there to update its contents, yet not to replace one file with another. If there is a one-to-one mapping of cached documents to dynamically retrieved documents, and that mapping is known by the cache manager, it can check to ensure that it is not overwriting the incorrect document. However, this still poses the problem of limiting the cache manager from dynamically adjusting the cache's contents to the current usage patterns of the Web server, since once a document is in the dynamic document cache, it can never be removed by the cache manager. A security tradeoff exists, therefore; by allowing the cache manager to delete data from the file system, it opens up the possibility of destroying data that is completely static, and therefore has method of dynamic generation in the case of a cache miss. However, as in any cache system, the cache manager must be trusted to correctly manage the migration of data.

8.6 Future Caching Issues

One issue that frequently troubles cache designers is that of data modification; while this problem has not been addressed in this thesis, it remains as a future consideration when designing a mixed retrieval caching system. What is the procedure for dealing with data that is modified while the data is in the cache? The cache management strategy must take this into account and ensure that all accessed data is valid.

One approach (the *write-through* approach) says that when a data modification occurs, the system should always write the modified value to the cache and also write that value immediately through to the primary data source. This method is simple to implement and ensures the permanent modification of the data's value, since it is written immediately to its primary storage space; however, frequent modifications to cached data slow down the system and somewhat defeat the purpose of having that data cached.

Another approach (the *write-back* approach) says to write the modified values only to the cache and set a *dirty bit* indicator that instructs the system to re-write the data back to the primary storage space only when the data is removed from the cache. While this approach is faster, therefore, it introduces the possibility that data modifications are not permanently stored in the event that the cache doesn't get a chance to write back the new value (for instance, if the system's power fails). Furthermore, it is more complicated to implement and requires a larger cache size to accommodate the dirty bits.

Numerous other approaches exist to deal with the problems associated with data modification. However, these issues do not need to be considered at the moment for the prototype implementation of mixed retrieval caching. Since this investigation deals with systems that are only serving data over the Web, the data can never be modified through the dynamic document cache; only in its original information system format can it be changed. If the system were to allow data modification over the Web also, a data modification doctrine analogous to one of those outlined above must be adopted.

Any such doctrine will invariably rely on the type of data being integrated and the methods of integration. However, in almost all cases it will probably be best to use a write-through approach to data modification, since by the nature of the problem, the Web will not be the only method of access to this data. If the data is accessed from the *other* information infrastructure, it will never first check the dynamic document cache and therefore never receive the modified data if was not immediately written through to its primary data source. Furthermore, not writing the data back to its primary source eliminates the possibility of run-time checks on the data itself that the primary system might perform when any of its data is modified. As a result, invalid data might result in the dynamic document cache if a write-back approach is taken. Finally, although the dynamic document cache represents the Web-represented content of the data, it might lose information in the publishing process; modifying the data in the cache might not "un-publish" in the desired manner back to the primary data source. Consequently, a write-through approach is recommended for future implementations of a mixed retrieval HTTP server.

9. Mixed Retrieval Server Performance

9.1 Comparison to Unmodified Server

When determining the performance of the mixed retrieval server (MRS), first it is necessary to compare its performance to the unmodified CERN server to determine how much of an overhead the modification had added. This was done by performing multiple, identical experiments on both the mixed retrieval server and the CERN server and comparing their response times.

The conditions in each test checked the server's performance under conditions in which a number of factors varied. The first variable was the retrieval method used by the server when returning the documents. Simple static and dynamic retrieval were the two cases, since the unmodified CERN server did not support mixed retrieval. In the case of the mixed retrieval server, they were simply static documents - not cached virtual documents. Similarly, for the case of dynamic retrieval, the mixed retrieval server was tested with straight CGI-script requests - not generated virtual documents.

The second variable used in the tests was the file size of the requested document. This was done to rate each server's performance as a function of document size, since larger files take longer to generate or return. The values used for this test ranged from 0 bytes (to test each server's retrieval overhead) to 50 kilobytes (to represent a large HTML document). These files were comprised strictly of unformatted ASCII text, and were identical in their static and dynamic versions. The CGI scripts that generated the text simply returned the specified number of characters.

These two variables were used to determine the relative performance of the mixed retrieval server when compared to the unmodified CERN server. Another factor was also varied - the number of documents requested - but since these tests were only geared at determining the average performance, the data for the most number of requests (1000) was used since more sample points provide more statistical significance.

These tests were conducted with the W3C Line Mode Browser [27], a text-based Web client that can be instructed to perform no formatting on the retrieved data. A shell script (see Appendix A: Test Conditions) automated the process by directing the browser to make the specified requests. Because of the long time delay between the start and end of trials for 1000 requests (sometimes on the order of twenty minutes), a granularity of seconds was deemed acceptable for calculating the average retrieval times. The UNIX date command was run at the beginning and end of each trial to determine the total trial time, then

this difference was divided by the number of requests to determine the average number of seconds per request.

Full results of these trials can be seen in Appendix B: Performance Data; the test conditions are detailed in Appendix A: Test Conditions. The conclusions from this data is summarized here. Comparing the mixed retrieval server to the unmodified CERN server shows that no additional overhead was added to the mixed retrieval server for either static retrieval or dynamic retrieval. Table 3: CERN Server Performance vs. Mixed Retrieval Server Performance summarizes these results; the average time per request does not vary significantly between servers for a given retrieval method and document size.

Document size	CERN server static retrieval (sec/req)	MRS static retrieval (sec/req)	CERN server dynamic retrieval (sec/req)	MRS dynamic retrieval (sec/req)
static / 0k	0.08	0.08	0.10	0.10
static / 2.5k	0.08	0.09	0.16	0.16
static / 5k	0.09	0.09	0.22	0.22
static / 25k	0.10	0.10	0.73	0.73
static / 50k	0.12	0.12	1.36	1.36

Table 3: CERN Server Performance vs. Mixed Retrieval Server Performance

9.2 Comparing Mixed Retrieval to Static and Dynamic Retrieval

In addition to comparing the mixed retrieval server to the unmodified CERN server, it is also important to compare the new retrieval methods in the mixed retrieval server to their CERN server predecessors to determine the overhead that mixed retrieval adds to a standard request. Specifically, the performance of retrieving cached virtual documents should be compared to that of retrieving a simple static document, and the performance of retrieving a generated virtual document should be compared to that of retrieving a dynamically-generated (standard CGI-script) document.

Measurements were taken for these values using the same variables and test methods as described above. Full results of these trials can also be found in Appendix B: Performance Data; Table 4: Virtual documents vs. Normal documents summarizes these results; the average time per request of a virtual document is not significantly higher than that of a normal document.

Retrieval method / size	Normal document (sec/req)	Virtual document* (sec/req)
Static / 0k	0.08	0.08
Static / 2.5k	0.09	0.09
Static / 5k	0.09	0.09
Static / 25k	0.10	0.10
Static / 50k	0.12	0.12
Dynamic / 0k	0.10	0.11
Dynamic / 2.5k	0.16	0.17
Dynamic / 5k	0.22	0.23
Dynamic / 25k	0.73	0.73
Dynamic / 50k	1.36	1.37

Table 4: Virtual documents vs. Normal documents

*In the case of virtual documents, "static" refers to a cached virtual document, and "dynamic" refers to a generated virtual document.

This data leads to the conclusion that the new retrieval methods do not add a significant overhead to the cost of retrieving a virtual document, regardless of whether it is cached or not. Specifically, a 0.01 second/request overhead is added only to non-cached (generated) virtual documents. In the worst case scenario of a 0k document, this represents a 10% retrieval time overhead:

% Overhead = (Added retrieval time / Normal Retrieval Time) * 100
= (0.01 / 0.10) * 100
= 10.0 %

For the more normal case of a 5k document, it only represents a 4.5% overhead. These values, however, summarize the worst-case scenario for a generated virtual document, since these tests were run with generators programmed specifically to have the lowest possible overhead. As a result, more typical generators will have a much higher overhead than the figures recorded for these tests. Therefore, this worst-case tradeoff in performance for dynamically-generated virtual documents will most likely be greatly improved in a more realistic application.

10. A Potential Application: Hierarchical Streaming

10.1 New Retrieval Mechanisms

Mixed retrieval opens up new possibilities for the way Web servers retrieve and generate data. While the basic features and applications of mixed retrieval were investigated in this thesis, new areas still exist that can be explored.

With the binding removed between a document's URL and its retrieval method, the Web server is no longer limited in the way it can retrieve data based on a request. Furthermore, dynamic document generators can also be bound to arbitrary parts of a URL. New data generation paradigms may be introduced in a mixed retrieval Web server that build upon these new features. One such system will be briefly discussed, although mixed retrieval opens up the possibility for others.

10.2 The Hierarchical Streaming Process

With support for virtual collections at arbitrary locations in the URL comes the potential for hierarchy in the naming of Web documents. Even though hierarchy can be used simply as an organizational technique for increasing the scalability of the system, it may also have implications about the data itself that is contained within that hierarchy. Knowing a document's location in the system's hierarchical organization may itself render new information. With mixed retrieval, this new information may be captured using *hierarchical streaming*.

Hierarchical streaming is a new data generation mechanism that enriches a Web document on the server side based on its location in a hierarchy of virtual collections. If a client requests a document that is contained within a hierarchy of virtual collections, the server streams this document successively through the generators for those virtual collections; cached virtual collections are overridden by their dynamic counterparts for streaming only. If they know how to enrich the data, they do so; otherwise, they simply pass the data through. The general scheme for hierarchical streaming will proceed with the following algorithm:

1) At the lowest level of hierarchy in the URL (starting with the referenced document itself), attempt to access the resource through its default retrieval type as registered in the server's configuration file (i.e., an HTML document may be retrieved from the file system, or a Lotus Notes virtual document may be generated. Similarly, if the lowest level is itself a collection, the collection might be generated by its registered collection generator). If the resource cannot be accessed, generate a <no-content> stream. For illustration purposes, call the returned data stream D.
2) If this is the highest level of registered hierarchy in the URL, check the MIME type. If so: if the stream is <no-content>, generate Error 404, Not Found; otherwise, stream the data back to the client.
3) If this is not the highest level of the hierarchy, determine the next highest level of hierarchy (H) above the current one. Stream the data D into the collection generator registered for H, and repeat the process from (1).

This scheme allows multiple levels of hierarchy within a URL. Consequently, it is possible to retrieve data not only from the document that the client is specifically accessing, but also to enrich that data from any data generation mechanisms in its hierarchy. For instance, consider a user's mail folder that he wants to access from the Web. His inbox is represented by the /Mail directory on his computer, which also happens to be a Web server. However, he also categorizes his mail under /Mail/Humor/, /Mail/Friends/, /Mail/Band/, and /Mail/Tech/, by moving the files to those directories from the inbox once they've been read. Collections can help organize this mailbox. The following lines in the server's configuration file set up the mixed retrieval mechanisms that will be used for the hierarchy:

```
AddCollection          /Mail/              categories.mailfolder
AddCollection          /Mail/*/            messages.mailfolder
AddType                .mail               <mail-type>
AddType                .mailfolder         <mail-folder>
AddGenerator           <mail-type>         MailToHTML.exe
AddGenerator           <mail-folder>       MailFolderToHTML.exe
```

With these additions, the user may currently access a list of Categories of his mailbox from the Web using a URL such as:

```
http://site/Mail/
```

This URL would be recognized by the server as the virtual collection registered in the first "AddCollection" line above. It would consequently retrieve the cached version (under "/Mail/categories.mailfolder") or generate the collection dynamically to produce a list of the mail categories.

Similarly, the user can access a list of messages in a category through a request such as:

```
http://site/Mail/Category1/
```

This URL would be recognized by the server as the virtual collection registered in the second "AddCollection" line above, and retrieve the list of messages (or even subcategories) in Category1.

With the addition of hierarchical streaming to this system, however, the possibilities for retrieving data become augmented by the fact that the data itself is organized in a hierarchy. Consider a request for a specific mail document such as:

```
http://site/Mail/Computers/msg1.mail
```

Tracing the algorithm, a mixed retrieval server with support for hierarchical streaming would first retrieve the msg1.mail virtual document, either through its cached version or through its generator. The results of this retrieval, however, would not immediately be sent to the client as the response; the server would check the request URL and pass this data to the collection generator for the lowest level of registered hierarchy. In this case, it would send msg1.mail to the generator for the /Mail/Computers/ collection, MailFolderToHTML.exe. This generator would be programmed to interpret this input and check its MIME type; given that it is of the type <mail-type> (which the generator programmer knows in advance will be a type of data that this collection should augment), it will append any enriching features to the data that the collection itself knows how to generate. For instance, it might take a simple mail document, add the name of its category, and provide links to related messages in that category. This information would not be available to the mail document generator itself, but it would be to the collection generator; by existing in a hierarchy under the collection generator for a mail category, therefore, the mail document itself is enriched. The process of hierarchical streaming continues until the highest level of registered generators is encountered, in which case the final result is sent back to the client.

Consider the more complex case of a Lotus Notes Web interface in which different collection generators are registered for server, database, and document lists. Furthermore, assume that Notes is running at a company's site that wants its logo appended to every one of its HTML documents that it serves. The following rules register the relevant generators in the server's configuration file:

```
AddCollection       /                 index.logo
AddCollection       /Notes/           index.svrl
AddCollection       /Notes/*/         index.ndbl
AddCollection       /Notes/*/*/       index.ndcl
AddType             .logo             <logo-type>
AddType             .svrl             <server-list>
AddType             .ndcl             <doc-list>
AddType             .ndbl             <db-list>
AddType             .ndc              <notes-doc>
AddGenerator        <notes-doc>       ndcgen.exe
AddGenerator        <logo-type>       logo.exe
AddGenerator        <server-list>     serverlist.exe
AddGenerator        <db-list>         dblist.exe
AddGenerator        <doc-list>        doclist.exe
```

A user decides to access a specific document stored in a Notes database through the URL:

```
http://site/Notes/Server/DB/doc.ndc
```

The flow of data under this configuration can be seen in Figure 14: Hierarchical Streaming Control Flow.

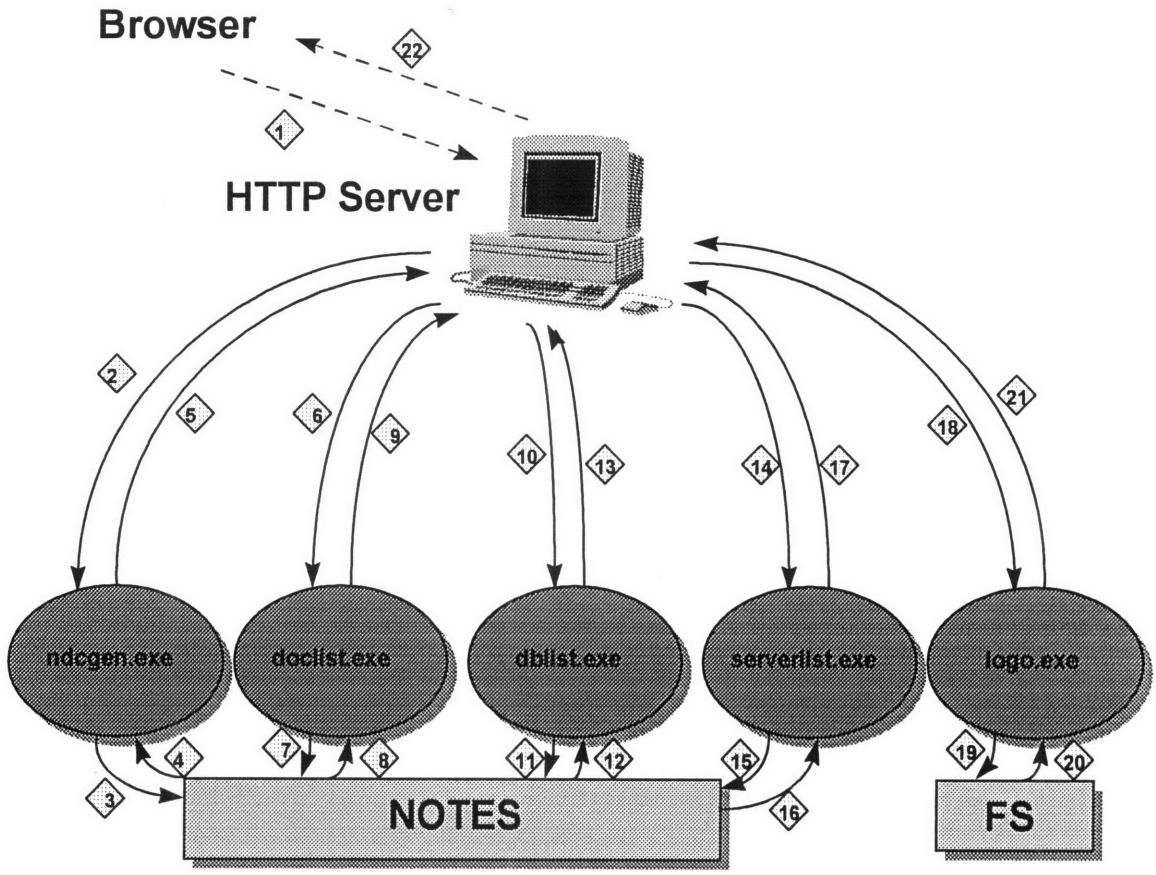

Figure 14: Hierarchical Streaming Control Flow

A client makes the initial request (1). The server cannot find doc.ndc on the file system, so it checks its bindings for the <notes-doc> MIME type. It determines that ndcgen.exe is the correct generator from the configuration, so it calls ndcgen.exe (2). Ndcgen.exe retrieves the data from Notes (3 and 4) based on the information in the URL (sent as a parameter to the generator) and returns it to the server (5). The server sees that the next level of registered hierarchy above the document is the DB/ collection, the type of which is bound to doclist.exe. This executable is run (6) and streamed in the result (5) of ndcgen.exe. Doclist.exe identifies this return type as a Notes document from its MIME type, and consequently only produces a partial list of documents around the selected document. This list is filtered into the return stream from ndcgen.exe by doclist.exe, and the total result is streamed back to the server (9). This process continues until the final stage in the URL hierarchy is reached at the root directory, which is registered

with logo.exe to insert a logo at the top of the page. If the final streamed result from the Notes calls happens to be a special MIME type that requires a Notes-specific helper-app, for example, logo.exe (as would all correctly designed collection generators) simply streams it through to the server without modification. However, if the Notes generators happened to parse their result into an HTML file or text file, for instance, logo.exe would most likely be encoded to insert the logo given those types of data streams.

10.3 Restrictions on the Use of Hierarchical Streaming

Hierarchical streaming cannot and should not be applied to all situations in which Web sites are organized into an apparent hierarchy of data. In this example, the /Mail/ collection gives some information about the data contained within it - specifically, that it is either a mail message or a mail category. Similarly, so does any category matching the pattern /Mail/*/. However, such a hierarchy not only organizes the type of data semantically, it also organizes its data *format*. Hierarchical streaming should only be applied to systems in which the return data type is *structured*, and in which collection generators can be built that can reliably make use of this structure. Virtual collections in a hierarchical streaming system should adhere to a standard set of design principles (and an API) to ensure that they will be able to interoperate. Specifically, they should:

1) Interpret the standard input stream and determine its MIME type.
2) Take appropriate action depending on this MIME type. Specifically, in all unknown cases it must stream the result through to the server without modification. It should be designed around the special cases that it is likely to encounter; for instance, a <doc-list> generator might interpret <notes-doc> and <no-content> differently; <no-content> might cause it to construct a Notes View, while <notes-doc> might cause it to construct a minimized Notes View.
3) In general, yield to the lower level of hierarchy unless they explicitly know how to handle the data type, since lower levels contain the most specific information requested.

10.4 The <no-content> Stream

If a document does not exist when the server initially attempts to retrieve it, the <no-content> stream is passed to the next highest collection level. This allows for the rare instance in which the collection generator itself would know to generate the document itself if no document can be found or generated. While it is not good practice to substitute such collection generators for virtual documents, it can apply in some situations in which very high-level collections encompass a very broad range of documents. In such a case, this collection functions mainly as a standard CGI script, accepting the request URL simply as a

parameter to direct its execution. The advantage that this gains, however, is that this collection can appear anywhere within the request URL, and higher levels of the hierarchy can still function as a normal hierarchical streaming system.

For instance, consider the case of a semantic file system [28] in which hierarchy represents a series of document properties rather than locations. Collections ending in ":" indicate a field, while others represent values for those fields. The web server does not have this information, however; a single collection generator is registered under the /sfs/ hierarchy that interprets the URL and generates the directories. A URL such as:

```
http://site/sfs/owner:/smith/text:/resume/
```

would be interpreted initially as a physical location, since no hierarchy is registered for /sfs/*/*/*/*/. However, since no such directory exists, a <no-content> stream would be passed to the next-highest level of registered hierarchy, /sfs/, the collection generator of which would interpret this stream, parse the URL, and generate the directory listing. It would seem to cause many problems if the URL actually referenced a document such as:

```
http://site/sfs/owner:/smith/text:/resume/bio.txt
```

However, a trace through the algorithm will show that this is not the case. Interpreting the .txt extension as a standard file to be shipped back to the client, the server will attempt to access the file /sfs/owner:/smith/text:/resume/bio.txt, which it will not find. Since the next level of hierarchy is /sfs/, its generator will be called with the <no-content> stream. As before, it will parse the URL; however, it will determine that a specific file has been requested and will act as a filename translator for the Web server; it will know how to locate that document in the semantic file system, and hence return that file to the server.

10.5 Conclusion

Hierarchical streaming is one new form of server interaction with data that is possible in a mixed retrieval system. The benefit of having virtual collection generators present at arbitrary points in a URL facilitates the enrichment of data that can be extracted from its hierarchical organization structure. Other such methods of data generation can be derived given the new features of a mixed retrieval server.

11. Related Work

Mixed retrieval and mixed retrieval caching focus on problems similar to those addressed in other areas of work.

In the areas focusing on integration between the Web and other information systems, a number of related developments exist. Varela, et al, discuss Database Browser (DB) [29], a system that creates server-side sessions with a back-end object-oriented database to increase the efficiency of Web access to that database. The system also provides for other services that help integrate it with the Web. Barta and Hauswirth discuss the problems involved in integrating the Web with legacy data systems based on terminal sessions, focusing on maintaining state over the stateless HTTP protocol [30]. Port, et al, focus on the facilities required for a company to deploy applications through the Web beyond their own enterprise [31].

Focusing on Web access to legacy information are Shklar and Shah [32], who discuss various methods of legacy data retrieval, including the tradeoffs of CGI-script filters, browser-native MIME types, and extracting meta-information for search and retrieval. OmniPort [33] is a system that allows a consistent access interface to heterogeneous legacy data.

Rowe and Nicholas address the issue of naming [34], specifically detailing the different modes of transparency which a Web-based name service should provide. Gifford, et al, address the concept of "virtual directories" in their discussion of semantic file systems [28].

Eichmann discusses the current data retrieval methods available when integrating the Web with another information system [35]. Nicol presents DynaWeb [36], an HTTP server that provides an interface to multiple information sources by transparently forking specific processes depending on the type of request.

In the areas of Web document caching, Abrams, et. al, discuss the design issues involved in proxy server caching [37]. Malpani, Lorch, and Berger discuss the problems of a single document cache and propose a method for distributing requests to multiple caching proxies [38]. Glassman provides an early exploration the properties of caching relays (proxies) [39].

Finally, an example similar to hierarchical streaming is presented by Brooks, et al, in which HTTP proxy servers modify the content of the data stream that they return to the client [40].

12. Conclusions

12.1 Results of this Thesis

This thesis focused on the problems of data retrieval on the Web when integrating it with another information system. Two main methods of data retrieval currently exists, *static retrieval* and *dynamic retrieval*; these methods have different benefits to the server and the system as a whole. These methods can be used to integrate the Web with another information system; however, they cannot be interchanged for a given document without either changing its URL or manually redirecting its request. Both of these alternatives were found to be impractical.

A mixed retrieval mechanism was designed to remove the binding between request URLs and the data retrieval mechanisms used by Web servers, and to allow for automatic redirection of specific requests on the server side. Mixed retrieval could then eliminate the problems of migrating documents between static and dynamic retrieval and hence allow a more seamless integration between the Web and another information system.

Conclusions can be drawn from the results obtained from the implementation of a reference mixed retrieval Web server. First, there are no technical limitations to including mixed retrieval in Web servers. The security of isolated CGI scripts can be maintained while removing the binding between document names and their retrieval methods. The added overhead incurred by mixed retrieval is negligible; performance of a mixed retrieval server is not noticeably worse than that of an unmodified server.

Web clients can benefit from mixed retrieval since it reduces a server's need to create dead links. Web site administrators benefit from it because it allows a site to become more manageable when that site serves data from another information system. The integration between the Web and that information system can be controlled more effectively without the need to worry about breaking links. Furthermore, in mixed retrieval caching, automated cache managers can dynamically adjust the data retrieval mechanisms depending on the current server load to increase the server's ability to handle large numbers of requests.

Mixed retrieval, however, is not the solution for every Web site that is integrated with another information system; while mixed retrieval allows these benefits, it also presents a number of issues that must be carefully considered depending on the type of data being served. The most pressing problem to be decided

is whether the data can be effectively partitioned to achieve the benefits of mixing static and dynamic retrieval, and if so, how a mixed retrieval server can determine these partitions.

12.2 Future Work

A number of issues regarding mixed retrieval need to be resolved before it can be deployed in a real system. While mixed retrieval caching was explored in this thesis as the decision-making process to determine virtual document retrieval methods, any number of other methods exist. These methods need to be explored and compared to mixed retrieval caching to determine the overall strategy. Furthermore, the methods for identifying and mapping virtual documents to their static and dynamic counterparts should be investigated. Throughout these studies, special consideration should be paid to new security holes that the system would open.

In the case of mixed retrieval caching, the main topic for future work is in development of cache management strategies that make sense in such a system. The details still must be resolved for the communication mechanism between the Web server, the Web client (including caching proxies), and the information system, in order to facilitate a practical cache manager.

Hierarchical streaming represents a new potential application for mixed retrieval because of the ability to have multiple dynamic generators bound in different locations to a single URL. Other new designs such as this should be explored based on this new functionality. The performance costs and practicality of hierarchical streaming should also be determined.

In general, though, this thesis focused on the issue of data retrieval from an information system when it is integrated with the World Wide Web. A great number of other factors need to be considered when determining overall how to integrate the Web with an existing information system containing legacy data. The issue of *representation* considers how the legacy data is to be represented on the Web. *Security* determines how the security mechanisms present in the primary information system will be transferred or accommodated through a Web interface. The issue of *input* determines how the integrated system will handle the modification of data in the primary information system through the Web. Therefore, while mixed retrieval addresses one specific problem involved in the integration of the Web and other information systems, a number of other pressing issues need to be resolved.

13. Appendix A: Test Conditions

13.1 Common Parameters

Two sets of performance tests were run; the following conditions were present in both sets of tests. The tests were conducted on an Intel Pentium-120 processor under the Linux operating system running kernel 1.2.13. The computer was connected through a 10 megabit/second Ethernet 10-baseT connection to the Internet.

The tests were performed through a tcsh shell script that automated 1000 successive Web requests for each document using the World Wide Web Consortium's Line Mode Browser, a text-based Web client. The data was retrieved using the -n and -raw directive, and piped to /dev/null to eliminate formatting and output overhead. A sample request from this script can be seen in Figure 15: Sample Performance Test Script Call.

```
date
  for loop2 in 0 1 2 3 4 5 6 7 8 9; do
    for loop1 in 0 1 2 3 4 5 6 7 8 9; do
      for loop in 0 1 2 3 4 5 6 7 8 9; do
        www -n -raw http://ned-brainard.mit.edu:8080/zero.htm > /dev/null
      done
    done
  done
date
```

Figure 15: Sample Performance Test Script Call

Thus, the measurements taken also included the processing time of the shell script and the client. However, all measurements were used to measure relative performance; this extra overhead does not change between the tests.

The test script and both servers (the CERN server on port 8080, and the mixed retrieval server on port 8090) were run on the same machine. No requests or responses went over the network, to increase the accuracy of the measurements and eliminate the added overhead of network transport. Because the servers were run on non-default ports, no other HTTP requests were received during the tests to affect the results.

All CGI scripts were written in C and compiled with gcc version 2.6. All requested documents were either in HTML format or plain text. Because of the client request, however, no formatting was performed on the requests once they were received.

13.2 Static vs. Dynamic Document Performance Tests

The first series of tests were run to determine the retrieval overhead of dynamic documents using the CGI script mechanism; these tests are detailed in Section 3.2: Methods for Legacy Data Retrieval. This was done by measuring the retrieval times of various sizes of documents retrieved both statically and dynamically. The content of the documents was the same, not depending on the retrieval method. To ensure that the CGI scripts were as efficient as possible (so as to verify that only the overhead of the CGI mechanism was being measured), they were programmed to simply send back the same document as the statically-requested file. This file was read into the script in blocks of 32,768 bytes according to the method outlined by Stevens [41] for maximum I/O efficiency.

13.3 Mixed Retrieval vs. Static and Dynamic Retrieval Performance Tests

The other series of tests simply were aimed at determining any difference in performance between the mixed retrieval server and the unmodified CERN server. For these tests, efficient CGI scripts were not needed, since only the difference in retrieval times between servers was relevant. These CGI scripts simply printed out the desired number of characters to produce a document of the requested size.

14. Appendix B: Performance Data

The following data was recorded under the test conditions in Appendix A: Test Conditions. These results were used to verify that mixed retrieval adds an insignificant overhead to the current performance of the CERN server. Chapter 9, Mixed Retrieval Server Performance, details these results.

Test Type	Size	Average	Sec/Req	Req/Sec	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Static - CERN	0	82.2	0.08	12.17	82	82	82	83	82
Static - CERN	2560	84.2	0.08	11.88	84	84	86	83	84
Static - CERN	5120	91.4	0.09	10.94	91	91	91	92	92
Static - CERN	10240	94.4	0.09	10.59	94	94	94	95	95
Static - CERN	25600	98.4	0.10	10.16	99	98	98	99	98
Static - CERN	51200	119.8	0.12	8.35	120	119	119	121	120
Dynamic - CERN	0	100.2	0.10	9.98	100	100	100	100	101
Dynamic - CERN	2560	161.8	0.16	6.18	162	160	162	162	163
Dynamic - CERN	5120	221.8	0.22	4.51	223	223	223	216	224
Dynamic - CERN	10240	349.2	0.35	2.86	349	349	349	349	350
Dynamic - CERN	25600	725.4	0.73	1.38	727	719	727	727	727
Dynamic - CERN	51200	1359	1.36	0.74	1359	1360	1359	1358	1359
Static	0	82.4	0.08	12.14	82	82	82	83	83
Static	2560	86.2	0.09	11.60	86	86	86	86	87
Static	5120	91.2	0.09	10.96	90	92	90	91	93
Static	10240	90.4	0.09	11.06	90	91	90	91	90
Static	25600	99	0.10	10.10	99	99	99	99	99
Static	51200	120	0.12	8.33	120	121	120	120	119
Dynamic	0	100.2	0.10	9.98	100	101	100	100	100
Dynamic	2560	162	0.16	6.17	162	162	162	162	162
Dynamic	5120	223.2	0.22	4.48	224	223	223	223	223
Dynamic	10240	347.4	0.35	2.88	350	341	349	349	348
Dynamic	25600	727.6	0.73	1.37	727	727	728	728	728
Dynamic	51200	1359.2	1.36	0.74	1360	1359	1358	1360	1359
Mixed - HIT	0	83	0.08	12.05	83	83	82	83	84
Mixed - HIT	2560	85.6	0.09	11.68	85	86	86	85	86
Mixed - HIT	5120	90.8	0.09	11.01	91	90	90	91	92
Mixed - HIT	10240	90.6	0.09	11.04	90	90	91	91	91
Mixed - HIT	25600	102.4	0.10	9.77	102	103	102	102	103
Mixed - HIT	51200	119.6	0.12	8.36	121	120	119	118	120
Mixed - MISS	0	105.8	0.11	9.45	106	106	106	105	106
Mixed - MISS	2560	168.2	0.17	5.95	168	168	168	169	168
Mixed - MISS	5120	228.4	0.23	4.38	229	228	228	228	229
Mixed - MISS	10240	355.4	0.36	2.81	355	356	355	355	356
Mixed - MISS	25600	731.8	0.73	1.37	732	731	733	732	731
Mixed - MISS	51200	1365	1.37	0.73	1365	1364	1367	1364	1365

Table 5: Performance Data - Measurements taken to compare the mixed retrieval server's performance to that of the unmodified CERN server.

15. Appendix C: Sample Server Configuration

The following is a sample configuration file (httpd.conf) used for the mixed retrieval server. Comments (delimited by the # character) mark the newly-added configuration rules.

```
# General setup options
ServerRoot /usr/src/WWW/server_root
Port       80
Caching On
UserId     nobody
GroupId    nogroup
AccessLog  /usr/etc/httpd.log
ErrorLog   /usr/etc/httpd.errors
LogFormat  Common
LogTime          LocalTime
UserDir    www
Welcome welcome.htm
Welcome top.htm


#
#   New encodings
#

AddType    .suff  text/suffix
AddType    .ngn   text/nothing
AddType    .virt  text/virtual
AddType    .thesistest  text/thesistest
AddType    .alb   text/album
AddType    .alc   text/album-collection

#used for performance test

AddType    .zro   text/zero
AddType    .fve   text/fivek
AddType    .fty   text/fiftyk


#
#       Generators
#
# Note:  These rules are NEW for the mixed retrieval server

AddGenerator     text/suffix  /generators/docgen  text/html
AddGenerator     text/nothing /generators/nogen   text/html
AddGenerator     text/virtual /generators/virtgen text/html
AddGenerator     text/thesistest     /generators/thesistest    text/html
AddGenerator     text/album   /generators/albumgen      text/html
AddGenerator     text/album-collection     /generators/albumlist
    text/html

#used for performance test

AddGenerator     text/zero    /generators/zero    text/html
AddGenerator     text/fivek   /generators/fivek   text/html
AddGenerator     text/fiftyk  /generators/fiftyk  text/html

#
```

```
#   Collections
#
# Note:  These rules are NEW for the mixed retrieval server

AddCollection    /testdir/    index.ngn
AddCollection    /virtdir/    index.virt
AddCollection    /testdir/level2/    index.virt
AddCollection    /virtdir/*/  index.ngn
AddCollection    /slowdir/    index.virt
AddCollection    /Album/      index.alc

#
#   Scripts; URLs starting with /cgi-bin/ will be understood as
#   script calls in the directory /usr/etc/cgi.  Similar for
#   /usr/etc/generators/
#

Exec       /cgi-bin/*          /usr/etc/cgi/*
Exec       /generators/*       /usr/etc/generators/*

#
#   URL translation rules
#

Pass       /*     /sorcerer/HTTP/*
```

16. Appendix D: Test Applications

To ensure that the server performed as specified, a number of small, self-contained unit tests were designed. The requests in Table 6: Test Results were determined to represent the possible retrieval cases that the server would encounter. Alongside each test case is a description of the expected result, and a listing of any deviations from this expected result in the actual test. It was found that each test case worked as specified, without any deviations from the expected results.

These tests were run to check not only the mixed retrieval functionality, but also the correctness of the interaction between the server and the virtual document generators. This interaction consists of launching the correct generator bound to the document/collection type, as well as passing parameters to those scripts through environment variables. The test scripts used were programmed to produce an HTML output of the environment variables that they received. By checking the environment variables passed to the scripts, it was determined that this interaction was properly functioning.

Request	Expected Result	Deviations from Expected
Static document	Document returned from file system.	None
Dynamic document	CGI script run and resulting document returned	None
Directory listing	File Not Found; (not supported in mixed retrieval server)	None
"Welcome" document*	Welcome document returned	None
Document that doesn't exist and is not registered as virtual	File Not Found	None
Cached virtual document	Document returned from file system	None
Non-cached (generated) virtual document	Document generated from script	None
Cached virtual collection	Document returned from file system	None
Non-cached (generated) virtual collection	Collection generated from script	None
Static document within a virtual collection	Document returned from file system	None
Multi-level non-cached virtual collection	Lowest level collection's generator produces document	None
Virtual collection whose corresponding directory does not exist	Collection generated from script	None
Wildcard non-cached virtual collection	Collection generated from script bound to wildcard	None

Table 6: Test Results

* A "welcome" document is a document that is automatically returned from a directory when that directory is requested with a trailing slash ("/"). For instance, "http://web.mit.edu/" might return the equivalent of "http://web.mit.edu/index.html."

17. Bibliography

[1] Zisman, Mike. "Lotus Notes Web Strategy." Lotus Communication Strategy Briefing, Dec. 13, 1995. <http://www.lotus.com/mediadv/zispart2.htm>

[2] "Lotus Notes and the World Wide Web." Lotus Communique, July, 1995. <http://www.lotus.com/corpcomm/3236.htm>

[3] "Common Gateway Interface." <http://hoohoo.ncsa.uiuc.edu/cgi/intro.html>

[4] "The Common Gateway Interface." <http://hoohoo.ncsa.uiuc.edu/cgi/primer.html>

[5] "Performance Benchmark Tests of Unix Web Servers Using APIs and CGIs," Shiloh Consulting. November, 1995. <http://home.netscape.com/comprod/server_central/performance_whitepaper.html>

[6] Berners-Lee, Tim, et al. "HyperText Transfer Protocol – HTTP/1.0" October 14, 1995. <http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html>

[7] "HyperText Transfer Protocol." < http://www.w3.org/pub/WWW/Protocols/>

[8] Daniel, Ron. "Uniform Resource Identifiers (URIs)." <http://www.acl.lanl.gov/URI/uri.html>

[9] Connelly, Daniel W. "WWW Names and Addresses, URIs, URLs, URNs, URCs." <http://www.w3.org/hypertext/WWW/Addressing/Addressing.html>

[10] Berners-Lee, Tim. "Uniform Resource Locators." <http://www.w3.org/hypertext/WWW/Addressing/URL/url-spec.html>

[11] "NSFNET Backbone Traffic Distribution by Service," April, 1995. <ftp://nic.merit.edu/nsfnet/statistics/1995/nsf-9504.ports>

[12] Berners-Lee, Tim, et al. "Uniform Resource Locator (URL)." RFC 1738, December, 1994. <http://www.w3.org/pub/WWW/Addressing/rfc1738.txt >

[13] "Rules in the Configuration File." Configuration File of W3C httpd. <http://www.w3.org/pub/WWW/Daemon/User/Config/Rules.html#Redirect >

[14] "A Beginner's Guide to URLs." <http://www.ncsa.uiuc.edu/demoweb/url-primer.html>

[15] "Common Gateway Interface." <http://hoohoo.ncsa.uiuc.edu/cgi/intro.html>

[16] Stein, Lincoln D. "The World Wide Web Security FAQ." Version 1.1.5, January 2, 1995. <http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html#Q30 >

[17] "The Common Gateway Interface." <http://hoohoo.ncsa.uiuc.edu/cgi/primer.html>

[18] "Lotus Notes and the World Wide Web." Lotus Communique, July, 1995. <http://www.lotus.com/corpcomm/3236.htm>

[19] Stein, Lincoln D. "The World Wide Web Security FAQ," November 30, 1995. <http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq>

[20] Sollins, K., and Masinter, L. "Functional Requirements for Uniform Resource Names." RFC 1737, December, 1994. <http://www.w3.org/pub/WWW/Addressing/rfc1737.txt >

[21] Borenstein, N., and Freed, N. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies." RFC 1521, September, 1993. <http://www.informatik.uni-bremen.de/docs_mosaic/rfc1521.txt >

[22] "Directory Browsing." Configuration File of W3C httpd. <http://www.w3.org/pub/WWW/Daemon/User/Config/Directories.html >

[23] Frystyk, Henrik. "W3C httpd." August, 1995. <http://www.w3.org/pub/WWW/Daemon/ >

[24] Frystyk, Henrik. "W3C Reference Library." December, 1995. <http://www.w3.org/pub/WWW/Library/ >

[25] "W3C httpd Administration." July, 1995. <http://www.w3.org/pub/WWW/Daemon/User/Admin.html >

[26] Berners-Lee, Tim, et al. "HyperText Transfer Protocol — HTTP/1.0" October 14, 1995. <http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html#Code404 >

[27] Frystyk, Henrik. "W3C Line Mode Browser." November, 1995. <http://www.w3.org/pub/WWW/LineMode/ >

[28] Gifford, David K., et al. "Semantic File Systems." 13th ACM Symposium on Operating Systems Principles, October 1991.

[29] Varela, C., et al. "DB: Browsing Object-Oriented Databases over the Web." Department of Computer Science, University of Illinois. World Wide Web Journal, Fourth International World Wide Web Conference; pg. 209. O'Reilly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers2/282/>

[30] Barta, Robert A., and Hauswirth, Manfred. "Interface-parasite Gateways." World Wide Web Journal, Fourth International World Wide Web Conference; pg. 277. O'Reilly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers/273/>

[31] Port, Graeme, et al. "Requirements for Taking Applications Beyond the Enterprise." World Wide Web Journal, Fourth International World Wide Web Conference; pg. 703. O'Reilly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers/319/>

[32] Shklar, Leon, and Shah, Kshitij. "Web Access to Legacy Data." Fourth International World Wide Web Conference, 1995. <http://www.cs.rutgers.edu/~shklar/www4/intro.html>

[33] Ford, Shelly G., and Stern, Robert C. "OmniPort: Integrating Legacy Data into the WWW." Second International World Wide Web Conference, July 14, 1994.

[34] Rowe, Kenneth E., and Nicholas, Charles K. "Reliability of WWW Name Servers." Third International World Wide Web Conference, April 11, 1995. <http://www.igd.fhg.de/www/www95/proceedings/papers/75/rowe_release_2/www-reliable.html>

[35] Eichmann, David. "Application Architectures for Web-Based Data Access." Fourth International World Wide Web Conference, 1995. <http://athos.rutgers.edu/~shklar/www4/eichmann.html>

[36] Nicol, Gavin Thomas. "DynaWeb: Interfacing large SGML repositories and the WWW." World Wide Web Journal, Fourth International World Wide Web Conference; pg. 549. O'Rielly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers/112/>

[37] Abrams, Marc, et al. "Caching Proxies: Limitations and Potentials." World Wide Web Journal, Fourth International World Wide Web Conference; pg. 119. O'Rielly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers/155/>

[38] Malpani, Radhika, et al. "Making World Wide Web Caching Servers Cooperate." World Wide Web Journal, Fourth International World Wide Web Conference; pg. 107. O'Rielly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers/59/>

[39] Glassman, Steven. "A Caching Relay for the World Wide Web." Systems Research Center, Digital Equipment Corporation. First International Conference on the World Wide Web, May 25, 1994. <http://www.elsevier.nl/cgi-bin/WWW94link/10/overview>

[40] Brooks, Charles, et al. "Application-Specific Proxy Servers as HTTP Stream Transducers." World Wide Web Journal, Fourth International World Wide Web Conference; pg. 539. O'Rielly & Associates, 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers/56/>

[41] Stevens, W. Richard. "Advanced Programming in the UNIX Environment." Addison-Wesley, 1992. Pgs. 55-57, 132.