Dynamic Documents: Extensibility and Adaptability in the WWW

M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

{kaashoek, pinckney, josh}@lcs.mit.edu

September 15, 1994

Abstract

We propose *dynamic documents* as an approach to extending and customizing the Mosaic interface to the WWW. Mosaic implements rigid policies that limit the range of environments in which it can function. Dynamic documents overcome this problem by allowing documents to implement their own policies. Dynamic documents are programs executed at a Mosaic client to generate a document; they are implemented as Tcl scripts. We have modified the NCSA Mosaic client to use a modified Tcl interpreter to run the dynamic documents it retrieves. The interpreter is designed to execute only those commands that do not violate safety.

We have used dynamic documents to configure an *adaptable Mosaic client* to a mobile environment. The use of dynamic documents allows adaptable clients to adjust to a variety of mobile platforms and network environments. We have used the adaptive client and dynamic documents to build an adaptive e-mail browser that employs application-specific caching and prefetching strategies. Both the browser and the displayed e-mail messages are dynamically customized to the computing environment in which they run.

1 Introduction

Information retrieval will be one of the driving applications of computing. Popular information retrieval tools need to be adaptable to many environments and platforms in order to allow users to have information at their fingertips. Recently, the World-Wide Web (WWW) and NCSA Mosaic have become extremely popular information retrieval tools [8, 5]. As currently designed, Mosaic's fixed policies and fixed interfaces prevent it from easily incorporating new applications or running on computing platforms differing from the high-performance workstations connected to high-bandwidth networks for which Mosaic was designed.

Currently, Mosaic implements rigid policies for fetching, displaying and interacting with documents. We propose to allow documents to set their own policies by using information that Mosaic provides about the client hardware and user preferences. The single set of policies tuned to the high-performance workstation environment would then no longer constrain the use of Mosaic in other environments. For example, Mosaic currently automatically displays all HTML documents it receives in response to a query. In the environment for which Mosaic was developed this policy performs adequately. However, on another platform–for instance, a mobile, wirelessly-networked computer–this display policy could easily overload the network and use all the limited screen space.

A number of groups are addressing this problem by extending the HyperText Transfer Protocol (HTTP) (e.g. Common Gateway Interface), the HyperText Mark-up Language (HTML), or by introducing new protocols (e.g. HTTP+) [2, 3, 1, 18]. Unfortunately, since protocols function as standards for information exchange they cannot be easily modified. We address this inflexibility by introducing *dynamic documents*. Dynamic documents are programs executed by an *adaptive WWW client* in order to produce information local to the client. The most common use of dynamic documents is to generate HTML that is displayed to the user.

Section 2 describes the use of dynamic documents to extend and to adapt Mosaic. Section 2.1 describes of the implementation of dynamic documents. Section 3 describes an adaptive Web client that supports dynamic documents, caching, and prefetching as well as how these features are used in an adaptive e-mail browser. Finally, we describe related work and conclude.

2 Dynamic Documents

We propose dynamic documents as an approach to extending and adapting WWW/Mosaic. Dynamic documents are programs fetched by a Mosaic client in exactly the same manner as any other MIME (Multipurpose Internet Mail Extensions) object. A dynamic document may contain both the data of the document and a set of scripts to control its presentation. Execution of a dynamic document may cause the client to perform a number of actions in order to generate a final presentation to the user; other documents may be fetched, new HTML may be generated, information local to the client may accessed, or an entirely separate interface may be generated. Thus, dynamic documents add a great deal of flexibility to Mosaic. Dynamic documents can be used to extend many Mosaic policies, however, this paper focuses on three main areas: information retrieval, information display, and interface processing.

All current proposals for altering client information retrieval policies rely on some form of built-in policy. These policies control when and how Mosaic retrieves and retains documents. In contrast, dynamic documents allow document authors to determine the retrieval policy on a case-by-case basis. For example, in Sections 3.1 and 3.2, we discuss adding client-side document caching and prefetching to Mosaic. The caching and prefetching modules do not have to use only one global policy. Flushing, invalidating, and prefetching policies can work completely correctly and apply document-specific knowledge to their decisions.

The choice of which information to display is closely linked to the choice of which information to retrieve. While both choices are rigidly set in the standard Mosaic client, dynamic documents allow both information display and information retrieval to be customized at the client. Currently, users may have a few configuration options available but these are session-specific and apply to all documents fetched. Dynamic documents expand these choices and apply them to each fetched document individually. For example, if Mosaic is running on a mobile computer with a small screen, a dynamic document can select some core subset of information to display to the user while creating links to others. Similarly, if the network bandwidth is limited, fetching large multi-media objects can be delayed and placed in the background while the user views more efficiently accessed information. Any location information available to the client can also be used to customize documents. Without involving the server, a client with a dynamic document might use location information to build a page with a map indicating the computer's current position and then compute a route to some specified destination. Further, as the client's operating parameters change (e.g. remaining battery power or available network bandwidth) the dynamic documents can alter their policies arbitrarily.

Mosaic's interface processing capabilities are even more rigidly set than its information retrieval and display policies. Currently, only simple interface actions, such as hyper-text link traversal and basic text searching, are performed locally. All other significant actions, such as processing the contents of fill-out forms and picture maps, are performed on the server. This increases network bandwidth consumption and limits the scalability of the WWW. In addition, there is no support for creating new interface elements. A document author is forced to use only what Mosaic provides.

We have lifted these restrictions through the use of dynamic documents. All interface processing can now be performed locally by Mosaic. When a fill-out form is completed, when a picture map is invoked, or when a key is pressed, the client can process the same data locally that was previously sent to the server and, like the server, produce a document as a result.

Dynamic documents can even be used to create an arbitrary interface in a separate window. Web interfaces are, therefore, no longer limited to what Mosaic can provide. If documents require more sophisticated input mechanisms or output forms they may create their own. Of course, they may also invoke any of Mosaic's built-in facilities. For example, a dynamic document can open a separate window, use Mosaic to fetch raw data from an HTTP server, and then create a 3-D graph of the data in the second window; alternatively, the dynamic document could simply provide a window that ran a user-interface while Mosaic was used to display data. This allows the document designer to trade the greater network bandwidth and server cycles required to generate and ship the 3-D graphical representation for the client processing power required to generate it on the client.

In designing dynamic documents, we decided to place a great deal of functionality at the client. The client must perform all document processing in addition to providing the Mosaic interface. Placing processing on the client side is inherently more scalable than forcing all processing to be performed on the server. We believe that this is the proper scheme since wireless network bandwidth is a more scarce resource than mobile computing processing cycles.

Other arrangements of dynamic document functionality are possible. For example, dynamic documents that run on a server could aid database queries. Dynamic documents could even partition themselves across the client and server dynamically. A database browsing document might decide that because of high server load and limited network bandwidth that queries should run on the server but all interface processing should be done on the client.

2.1 Implementation Issues

Dynamic documents are implemented as Tcl scripts that are executed upon receipt from a WWW server. Mosaic handles scripts as a new MIME type. Every document retrieved via HTTP has an associated MIME type that specifies the format of the data contained in the document. For each format, Mosaic has an interpreter, either internal or external, that is capable of presenting a view of the data. The "viewer" for dynamic documents is the Tcl interpreter.

Dynamic documents function by means of one or more callbacks. A callback is a procedure provided by the dynamic document and registered with the Mosaic client. It is invoked by the client when a particular condition is met. For example, when Mosaic needs to decide what information to display to the user, a callback may be invoked. This callback is then responsible for producing HTML that Mosaic displays.

One of the largest difficulties with dynamic documents is that they may be too powerful. The variety and complexity of possible dynamic documents raises three issues: safety, portability, and authoring. One can address the issues of safety and portability by carefully selecting the scripting language in which to implement dynamic documents. A variety of scripting languages are available from which to choose. We selected Tcl for a number of reasons. Currently, it is portable across Unix platforms, a widely known, safe version is available, it is easily extended, and, through the use of Tk, arbitrary X-based interfaces can be provided. In addition, the new Tcl/Tk project at Sun Microsystems promises to extend Tcl support to Macintosh and Windows environments, provide a secure execution environment, and provide an interactive graphical user interface builder [17].

For our prototype, we ensure safety by use of an interpreter that limits the operations a down-loaded script may perform. However, we could easily switch to Safe-Tcl and thereby greatly decrease the chance that we have left any security holes in the language [4].

Since dynamic documents are so general, we are investigating the possibility of assisting authors by automatically generating certain types of scripts. For example, it is possible to process standard HTML documents into scripts that restrict the amount of data initially transmitted to the client while fetching the rest in the background.

While dynamic documents add a great deal of flexibility to Mosaic, they do not entirely eliminate the need to extend protocols in Mosaic. In order to add functionality to the client, we need to define the primitives available to be invoked by dynamic documents. These primitives define the ways in which dynamic documents find out about and influence the rest of the world. While many primitives are possible, we have decided only to address the issues of flexibility in information retrieval, information display, and interface processing. Thus, we provide functions to fetch information from HTTP servers, to query parameters of the local computer, and to perform user-interface actions, as well as functions to control the document cache. We also need to specify which parts of Mosaic invoke which callbacks, which parameters are passed in those callbacks, and what side effects of the callback's execution are expected.

3 An Adaptable WWW Client

We have designed and implemented an adaptable WWW client based on NCSA Mosaic that supports dynamic documents. Using dynamic documents we can configure the client to run efficiently on a number of platforms. As an example, we have used dynamic documents to adapt our client to a mobile platform. In particular, we have adapted and extended the client's document fetching policies and user interface. Additionally, we have implemented client side caching and prefetching as well as augmented Mosaic to generate traces of its caching and prefetching behavior. We have done this for two reasons: First, we hope to improve overall interactive performance, even in the presence of low-bandwidth networks. Second, caching and prefetching are two areas where document specific policies can be much more effective than fixed global policies and so they provide good examples of the use of dynamic documents.

We have implemented the full functionality of dynamic documents by modifying NCSA Mosaic version 2.4. The client runs on a stationary SPARCstation 10 running Solaris 1.0. By running it on a stationary computer we have been able to attract a user community to test out the client and to collect traces on the usage of dynamic documents, caching, and prefetching. We are using these traces to compare policies. We are currently porting the client to IBM ThinkPads running Linux and connected by a 1M-bit/s WaveLAN.

In this section we will discuss the implementation of caching and prefetching, as well as an example application—an adaptive e-mail browser—that employs dynamic documents to extend the interface of Mosaic and for application-controlled caching and prefetching.

3.1 Caching

Caching is an effective technique to improve interactive performance. In Mosaic we have implemented a document cache at the HTTP level. In a wired environment, the document cache is probably best stored at a proxy server so that it can be shared by an entire group of users. This architecture benefits from the locality due to the common interests and communications among users sharing the proxy server. In contrast, wireless connections often have low variable bandwidths that are less reliable. It is unclear where to place a proxy server for a mobile user who may roam among several wireless connections. Thus, the cache in a wireless environment can be better placed at the mobile computer itself. We have chosen to make the cache persistent across sessions in order to benefit from locality between Mosaic sessions. Additionally, a persistent cache may be pre-loaded for disconnected operation, as in the Coda file system [14].

We have implemented caching at the HTTP level for ease of implementation. When an HTTP request is made for a document that is not currently cached, the document is requested from the server, and an exact copy of the server's response is recorded. Later, when requests for this Uniform Resource Locator (URL) are made, the cached data is simply replayed as if it had come from the network layer [6]. When the document cache is full, a randomly picked entry is discarded.

Since WWW servers do not provide callbacks, stale cache data cannot be detected. To attempt to maintain the correctness of cached data, entries are cached for only a document-specific period of time. The period of time used is an estimate of the time until the next modification. The estimate is made by assuming that the time between when a document is fetched from a server and when it was last modified on that server is proportional to the time until the next modification will be made. We provide dynamic documents with the additional ability to explicitly flush items from the cache. This is the same basic method used by Glassman for the caching relay server [10].

This global caching policy is adequate. Several users have used our modified Mosaic on a SPARCstation 10 during the three-week period of our trace and have experienced a cache hit rate of about 40%. The document cache is secondary to the standard Mosaic image cache and the Mosaic history cache. Thus our caching statistics only reflect the load presented to our document cache. All other references for images are first handled by the image cache. Thus, the three caches may combine to display a hit ratio higher than that measured at the level of the document cache.

However, the global caching policy is not always correct. At times stale data is used while at other times data is needlessly fetched. We solve this problem by allowing the cache to be controlled by dynamic documents. A dynamic document can register a callback with the cache subsystem. This callback will be used by Mosaic to determine if the cached entry is fresh or stale. If the callback returns that the entry is stale, Mosaic will fetch the document again from the server. Otherwise, the cached copy will be used. Dynamic documents can also invalidate particular entries, forcing the documents to be fetched from the server the next time they are referenced. This arrangement allows dynamic documents to implement their own cache coherency protocols. The adaptive e-mail browser described in Section 3.3 below uses this mechanism to implement a caching policy that only fetches data when it has changed on the server.

3.2 Prefetching

To hide the latencies of slow, wireless links, we have implemented document prefetching. Documents are prefetched by forking a copy of Mosaic, performing a document fetch in the forked copy, and loading the fetched document into the cache. If a user requests a document while prefetches are in progress, the prefetches are suspended so that the full network bandwidth may be given to the user's request. After the user's request completes, any suspended prefetches are allowed to continue. The default prefetching strategy is to prefetch documents based on the user's Mosaic history list. Every time a user visits a page, Mosaic will begin prefetching all documents from the current page that the user has visited in the past.

This implementation suffers from two distinct problems. First, forking an entire process merely to perform prefetching is expensive and inefficient. Unfortunately, since the current Unix implementations of Mosaic are single-threaded, we are forced to use this approach.

Future versions of Mosaic promise to lift this restriction and thus allow much more efficient prefetching.

Second, and more importantly, this strategy generates relatively few future cache hits while consuming a considerable amount of network bandwidth. Our traces shows that, on average, 0.5 objects are prefetched for each object explicitly fetched by the user. Of these prefetched objects, only about 2% are actually used in the future. The others just waste bandwidth and unnecessarily load servers. This is clearly a very poor policy.

We are investigating better strategies. One alternative is to have the HTTP server monitor access patterns and then periodically augment documents with information that specifies what documents a client should prefetch, given that that client has requested a particular document.

Dynamic documents allow global prefetching policies to be replaced by document-specific ones. Prefetching is only effective when future access patterns can be determined. Additionally, prefetching uses network bandwidth so the penalty for prefetching a document that is not needed may be large. Since scripts have detailed document specific knowledge they are sometimes best equipped to make prefetching decisions. Our adaptive e-mail browser provides its own prefetching strategy in order to prefetch only those documents which it knows are likely to be used in the near future. This drastically reduces the number of objects prefetched while at the same time greatly increasing the chances that the prefetched objects will be viewed.[1] However, it is still unclear how prefetching in Mosaic will affect other networked applications, especially in a wireless environment.

3.3 An Adaptive E-mail Browser

To investigate the potential of dynamic documents, we have built an adaptive e-mail browser in which MIME-encoded e-mail is accessed through a Mosaic client. Dynamic documents in this application are used to dynamically generate the browser interface and views of e-mail messages. In addition, dynamic documents are used to implement application specific caching and prefetching strategies. This application did not require any changes to the Mosaic client itself beyond those needed to support dynamic documents.

The adaptive e-mail browser is organized as follows. The browser is implemented as a Tcl scripts contained in dynamic documents retrieved from an HTTP server when the user opens the URL for the browser. The dynamic documents then run on the mobile computer in order to generate the browser's interface. The browser's interface runs entirely on the client so that button clicks and typing do not result in messages to the server. Environment variables (e.g., screen size and network speed) determine which parts of e-mail messages the browser retrieves. At the client, the browser then generates an HTML page for the message to display. This setup dynamically moves the application interface from the server to the client. As a result, we reduce the load on the server, we reduce the network traffic between the client and server, and we improve interactive performance.

The browser adapts to the screen size and to the network speed of the platform on which

[1]The browser only prefetches objects referenced in the currently displayed mail message.

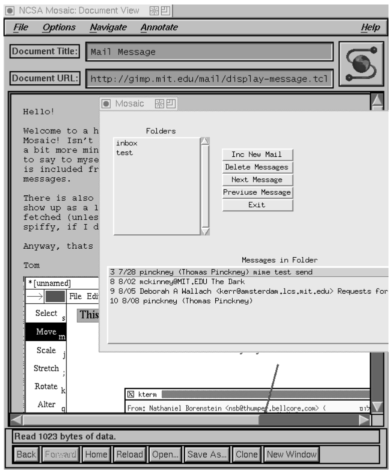
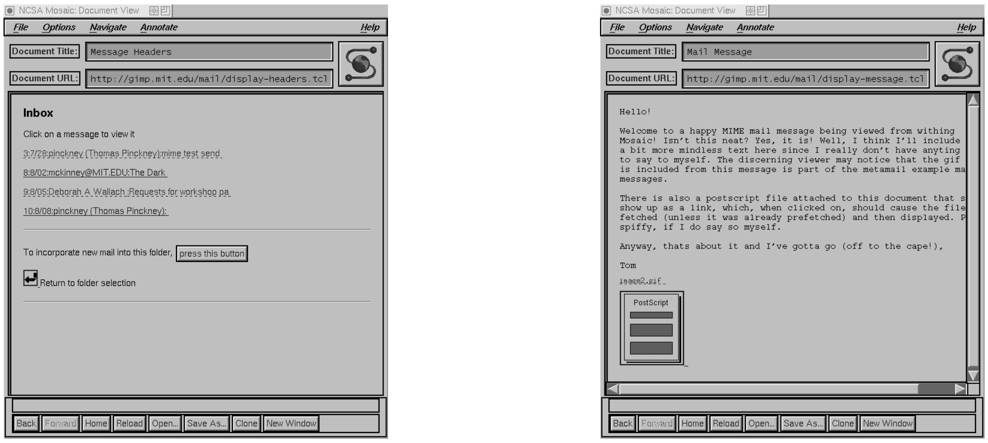


Figure 1: With a large screen and fast network, the browser displays the messages with text and graphics, and has an elaborate user interface in a separate window.

it runs. Depending on the screen size, it either displays a menu of URLs that point to individual mail folders and messages, or it generates a more elaborate interface in a second window that is more user friendly. In addition, messages adapt to screen size and network capabilities. For example, if the screen space is limited or the network speed slow, the browser generates links to pieces of the message rather than actually fetching and displaying the data completely.

Figures 1 and 2 illustrate how the browser adapts to different environments. Figure 1 shows the browser for a platform with a big screen and a fast network. In this configuration, the browser presents a specialized X interface in a separate window. All text and GIF objects in the message are displayed, while links are created to other objects (e.g., PostScript and other attached binary files). Figure 2(a) shows how the browser adapts its interface to a mobile environment. It has only one window with a more primitive interface based on the interface elements that Mosaic provides. However, as much interface processing as possible is done locally. Figure 2(b) shows the display of a mail message in the mobile configuration of the browser; only the first text object in the message is displayed while links are created leading to all other objects contained in the message. The important point about these figures is that the browser and mail messages adapt by using dynamic documents and client-side environment parameters. No changes are needed to the mobile Mosaic client or the served dynamic documents.

The browser employs caching and prefetching strategies based on the objects that are included in e-mail messages. The browser understands the MIME standard for bundling and transporting sequences of multi-media objects in a normal RFC822 compliant mail message [9]. If the browser decides to create a link to an object rather than including the object directly into the display, the browser may prefetch some of the objects in the background while displaying other parts of the message. In this way, the fetching of potentially large objects (e.g., GIF or PostScript images) can be overlapped with the user reading text. The browser can do a better job at prefetching than an application-independent scheme, as the dynamic documents can take advantage of knowledge about the data being displayed (e.g,



(a) (b)

Figure 2: The user interface (a) and the message content (b) are adapted to a mobile environment; only text and links are displayed.

text, GIF, PostScript) and of knowledge about how the application works.

Just as prefetching is under document control, so is cache invalidation. The browser controls which of its code and data pieces are cached and invalidated since only it has complete knowledge about when documents have changed. For example, the contents of different mail folders are cached and only invalidated when the user deletes mail from or incorporates new mail into the folder. This is vastly better than relying on a general caching and invalidation policy built into Mosaic. We have tried to quantify the performance improvement dynamic documents provide by looking at the caching and prefetching behavior that a normal user would generate. We assume that the cache is warm, which is a reasonable assumption since the cache is persistent and mail is a frequently used application. We also assume that a typical use of the e-mail browser will be to incorporate a small number of new messages and then to view them. Messages will be composed of at least text and maybe one or two attached images or PostScript objects. We traced using the browser under such conditions and found that the cache hit rates goes up to 60-70% from the previous 40% achieved using a global policy. In addition, the number of prefetches drops drastically, corresponding to the number of images included in our sample e-mail messages, all of which have a fairly high chance of being used. We do not intend this usage scenario, and associated numbers, to be taken as a general characterization of dynamic-document-controlled caching or even of the e-mail browser. It is only the measurement of the effects of one policy on one example workload and is intended to be just an illustration of what is possible.

4 Related Work

Using scripts for extending and customizing applications is not a new idea. Many examples of this approach can be found. A relatively successful example is PostScript [13]. Other examples include the Andrew Mail system, in which messages can contain fragments of LISP code that are executed at the receiver. This approach has been adopted for some more recent mailers that defined a safe subset of Tcl to do flexible mail processing. Recent work uses scripting languages to implement intelligent agents [7].

A number of projects have been extending the applicability of the WWW/Mosaic. In particular, Houh et al. implemented capabilities for shipping code from the client to the server to implement, for example, queries [12]. Mallery discusses an implementation of an HTTP server in Common Lisp and argues for dynamically moving functionality from servers to client using fragments of Lisp code [16]. Our work differs from this work in that we propose dynamic documents as a general approach for customizing and extending the WWW/Mosaic, and apply them to a mobile computing environment.

In the mobile arena, a number of groups have been working on Mosaic clients (or other user interfaces) for mobile computers. Most of this work focuses on the question what part of the client to run on the mobile computer and what part to run on a stationary computer [11, 20]. Landay and Kaufmann have used Scheme to ease the implementation of a split user interface [15]. Our work currently assumes that clients have enough processing power to run the client completely on the mobile computer. In the future we are planning to investigate how to implement dynamic documents on smaller mobile computers, and use Tcl scripts to split the interface, which should work particularly well for Mosaic clients that are being written using Tcl. Schilit et al. propose dynamic environment variables for customizing mobile applications while they are moving around; this idea could easily be incorporated into dynamic documents [20].

Caching and prefetching have been used for mobile computers. In particular, the Coda project has worked on making a file system that runs well on mobile computers [14, 19]. One of the potential advantages that dynamic documents offer is that they allow document-specific caching and prefetching. Such domain knowledge might reduce bandwidth consumption and might increase the number of cache hits. Glassman reports on a relay server that caches documents and gives a detailed analysis of its performance [10].

5 Conclusions

We have described dynamic documents, a mechanism for giving documents control of important policy decisions in Mosaic. Instead of relying on static policies built into Mosaic which must suffice for every document that Mosaic displays, we allow each document to implement its own policy. Among other things, dynamic documents allow Mosaic to adapt easily to different computing environments.

We have implemented an adaptable WWW client, based on Mosaic and for use in a mobile environment, that incorporates dynamic documents, caching, and prefetching. We have used dynamic documents to implement a mobile e-mail browser on this client that gracefully adapts to mobile computing environments. In addition, the browser uses application-controlled caching and prefetching strategies.

Our initial results based on the usage of the adaptable client on stationary computers for three weeks indicate that dynamic documents are a powerful approach to customizability and extensibility. We have used application specific information with dynamic documents to customize information display and interface processing. At the same time, we dynamically move interface processing to the client, decreasing network and server load and increasing scalability. We have also been able to guarantee correctness of a cache while dramatically increasing its hit rate and at the same time making prefetching feasible.

Acknowledgments

We thank David Gifford and James O'Toole for suggestions and ideas in the initial phase of this project. We thank Dawson Engler for suggesting HTTP server annotation of documents to support prefetching. We also thank Deborah A. Wallach, Wilson Hsieh, and Sanjay Ghemawat for their comments on versions of this paper.

About the Authors

M. Frans Kaashoek is an assistant professor at M.I.T. He is a member of the Laboratory for Computer Science (LCS) and heads the Parallel and Distributed Operating Systems (PDOS) group. His research interests include operating systems, parallel and distributed computing, networking protocols, computer architecture, programming languages, and compilers. He has published over 30 scientific papers and codeveloped a number of systems that are commercially or publicly available. In particular, he is one of the principal designers of the Amoeba distributed operating system and the Orca distributed shared memory system.

Joshua A. Tauber is a graduate student at M.I.T. and is a member of the PDOS group at the LCS. Before joining PDOS, Mr. Tauber was a software engineer with the Matsushita Information Technology Laboratory. His principal interest is in distributed computing systems. His most recent work focussed on system software for mobile computers. He developed approaches to provide consistent service as a computer roams among various resource domains.

Tom Pinckney is an undergraduate computer science student at M.I.T and is a member of the Parallel and Distributed Operating System group. He worked on a novel storage system for workstations, implemented dynamic documents for Mosaic and now spends his time dealing with classes.

References

[1] Common Gateway Interface. *http://hoohoo.ncsa.uiuc.edu/cgi/intro.html.*

[2] HTTP: a protocol for networked information. *http://info.cern.ch/hypertext/WWW/-Protocols/HTTP/HTTP2.html.*

[3] HyperText Markup Language (HTML). *http://info.cern.ch/hypertext/WWW/-MarkUp/MarkUp.html.*

[4] MIME extensions for mail-enabled applications: application/Safe-Tcl and multipart/enabled mail. *ftp://thumper.bellcore.com/pub/nsb/st/safe-tcl.ps.*

[5] Mosaic. *http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html.*

[6] Uniform Resource Locators. *http://info.cern.ch/hypertext/WWW/Addressing/URL/-Overview.html.*

[7] Intelligent agents issues. *CACM*, 37(7), July 1994.

[8] T. Berners-Lee, R. Caililiau, A. Luotonen, H.F. Nielsen, and A. Secret. The World-Wide Web. *Comm. ACM*, 37(8):76–82, Aug 1994.

[9] David H. Crocker. Standard for the format of ARPA internet text messages. *RFC 822*, Aug 1982.

[10] S. Glassman. A caching relay for the World-Wibe Web. In *Proc. First International World-Wide Web Conference*, pages 60–76, Geneva, May 1994.

[11] D. Goldberg and M. Tso. How to program networked portable computers. In *Proc. Fourth Workshop on Workstation Operating Systems*, pages 30–33, Napa, California, Oct. 1993.

[12] H. Houh, C. Lindblad, and D. Wetherall. Active pages. In *Proc. First International World-Wide Web Conference*, pages 265–270, Geneva, May 1994.

[13] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, Reading, MA, 1985.

[14] J. J. Kistler and M. Satyanarayananan. Disconnected operation in the Coda file system. In *Proc. Thirteenth Symposium on Operating System Principles*, pages 213–225, Pacific Grove, CA, Oct 1991.

[15] J.A. Landay and T.R. Kaufmann. User interface issues in mobile computing. In *Proc. Fourth Workshop on Workstation Operating Systems*, pages 40–48, Napa, California, Oct 1993.

[16] J.C. Mallery. A Common LISP hypermedia server. In *Proc. First International World-Wide Web Conference*, pages 239–247, Geneva, May 1994.

[17] J.K. Ousterhout. Usenet comp.lang.tcl posting. Sep 1994.

[18] Dave Raggett. A review of the HTML+ document format. In *Proc. First International World-Wide Web Conference*, May 1994.

[19] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile environment. In *Proc. Symposium on Mobile and Location-Independent Computing*, pages 11–28, Cambridge, MA, Aug 1993.

[20] B. N. Schilit, M. M. Theimer, and B. B. Welch. Customizing mobile applications. In *Proc. Symposium on Mobile and Location-Independent Computing*, pages 129–138, Cambridge, MA, Aug 1993.