INTERMEDIARIES: NEW PLACES FOR PRODUCING AND MANIPULATING WEB CONTENT

Rob Barrett Paul P. Maglio
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
{barrett,pmaglio}@almaden.ibm.com

Abstract

We propose a new approach to programming web applications that increases the web's computational power, the web's flexibility, and web programmer productivity. Whereas web servers have traditionally been responsible for producing all content, intermediaries now provide new places for producing and manipulating web data. We define intermediaries as computational elements that lie along the path of a web transaction. In this paper, we describe the fundamental ideas behind intermediaries and provide a collection of example applications. We also describe WBI, an implemented architecture for building intermediaries that we have used to construct many applications, including personal histories, password management, image distillation, collaborative filtering, targeted advertising, and web advising.

1. Introduction

The current model of the web gives the web server the full responsibility and privilege of defining the information that the browser will display. Displayed information may include text, images, dynamic pages and interactive applications. Yet one entity, the server, maintains complete control over the content that is delivered to the browser. Therefore, the server is the only place for computing content. In this paper, we describe how to enhance the computational power of the web by extending content computation beyond the server. We define *intermediaries* as computational elements that lie along the path of web transactions. They may operate on web data as the request is sent from the browser, passes through firewalls and proxies, is satisfied by the generation of a document, and as the document is returned to the browser. Intermediaries have access to web data at all these points, and are able to observe, respond to requests, and modify both the request and the resulting documents.

Intermediaries can be differentiated from two other forms of non-server computation, applets and browser plugins. Applets allow computation to be performed on the client, but the actual executed code is determined by the server that delivered both the references to the code and the code itself. Seen another way, applets affect only the particular page on which the server chose to place them. Browser plugins render new content types or replace default rendering engines, but do not allow a browser to control the content that is rendered. In this case too, the server controls the content.

Intermediaries represent a new model for programming web-based applications. In this model, the usual case of a server delivering a document to a browser is but one type of web transaction. An intermediary running on the client may respond to the browser request directly, producing personal, web-accessible applications for a particular user. An intermediary running on a workgroup server may respond to requests, producing a collaborative web-space. In addition to merely producing documents, intermediaries can modify documents produced elsewhere, allowing annotation, personalization, highlighting, and the integration of information from multiple sources. Intermediaries are a new way of thinking about web applications that provides much greater

freedom for design and implementation than traditional server-based web programming.

Many applications fit the intermediary model of web programming. Consider just a few:

Web Personalization

Our work on intermediaries was motivated by an interest in personalizing the web, that is, in changing the web's behavior depending on the particular user and that user's history of browsing [1]. Based on explicit instructions or on observations of a user's browsing, intermediaries can formulate a model of the user and then alter what the user sees to personalize the browsing experience. For example, pages can be annotated with user notes or customized links, alerts to related web pages can appear, or new resources, such as personal search engines, can be added.

Document Caching

Document caching, a common feature of current web browsers, can easily be cast in the intermediary mold. If the caching function is separated from the browser and implemented by an intermediary, the browser becomes simply a URL requester. In this case, the intermediary cache checks its store of pages to see whether the request can be satisfied locally or whether it must be sent to another intermediary or to the server. Figure 1 illustrates this application. One benefit of this approach is that the cache can be shared among multiple browsers or multiple people. Another benefit is that different caching strategies can be implemented without affecting the rest of the system.

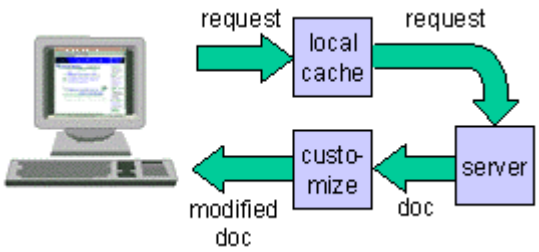

Figure 1. A request is issued from the browser to a caching intermediary. If the requested document is not in the cache, it is forwarded to the server intermediary, which produces the document. The document passes through a customization intermediary that personalizes the document and then returns it to the browser.

Content Distillation

Another obvious role for intermediaries is in content distillation [2]. The idea is that the same content can be represented at different levels of detail. For example, images can be stored at different resolutions and color depths, or with different degrees of lossy compression. A distilling intermediary transforms the original content obtained from the server to optimize for transmission speed, browser display capabilities, or browser computational limitations. Slow dial-in connections provide a compellingreason for investigating such schemes. The increasing popularity of devices with small displays, such as the PalmPilot, provide another.

Protocol Extension

The simplicity and power of the HTTP protocol is responsible for much of the web's success. With simplicity, however, comes limitations. Intermediaries allow many protocol extensions without altering deployed browsers and servers. For example, HTTP is not well-suited for long-latency wireless links. An intermediary can translate an HTTP transaction into a more efficient protocol before delivering it across a wireless link to another intermediary that transforms it back to HTTP. Likewise, new security structures or other enhancements can be added to an existing protocol simply by using a pair of translating intermediaries.

These examples illustrate how intermediaries add new computational power to the web. Because intermediaries

allow for computational processes at all points along the stream of a transaction, and because these processes can be dynamically reconfigured for each particular transaction, the web becomes customizable and extensible far beyond familiar scenarios. Moreover, the intermediaries approach breaks the servers' monopoly on producing web content. In what follows, we present our framework for intermediary-based computation in detail. First, we show how the familiar view of the web can be transformed by the introduction of the intermediary concept. Second, we describe our general architecture for composing intermediary functions from basic building blocks. Third, we present details of our implementation of this architecture. And finally, we discuss related work and the future of our approach.

2. The Move to Intermediaries

The intermediary approach is derived from the common browser-proxy-server model. In conventional thinking, the function of the browser is to initiate a request via an HTTP (or other protocol) connection to a proxy (or internet service provider). Alternatively, we can think of the browser as issuing a conceptual request that is handled by an implicit intermediary. The implicit intermediary handles the request by making the appropriate HTTP connection for the browser (see Figure 2). In making this intermediary function explicit, a new place for adding programmability appears. For instance, this client-side intermediary could choose to satisfy the request immediately from a cache, execute some local code to produce a dynamic page, access remote resources through another protocol, or simply make the appropriate HTTP connection as browsers normally do. By splitting the issuing of the request from the making of the HTTP connection, we have increased the flexibility of the web.

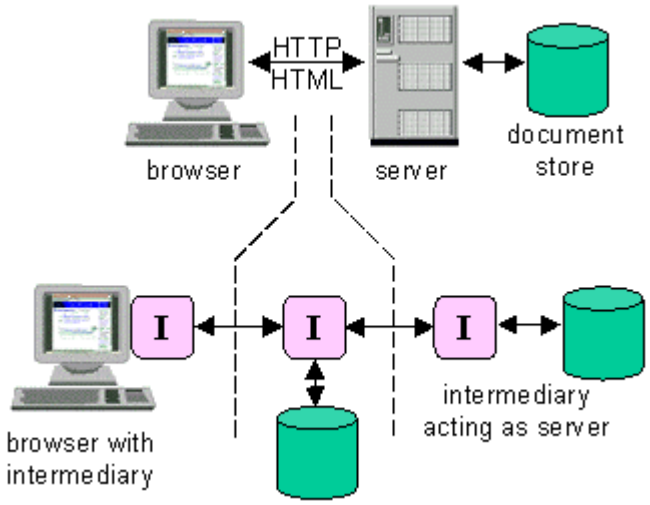

Figure 2. In the standard web model, the browser connects to a server and retrieves a document from its store. This connection is done through an "implicit intermediary" which takes the requested URL and retrieves the document from the server. We extend this model by adding fully programmable intermediaries ("I") at the client and mid-stream. The server is recast as an intermediary that produces documents in response to requests, though it could also continue forwarding the request to other intermediaries.

Now consider the proxy. Normally a proxy transparently transmits a request to the appropriate server and returns the resulting document. Some proxies add simple functions such as caching. But there is no reason why the proxy cannot be a full-fledged intermediary with a configurable response. In this case, some requests might simply be passed transparently, but others might access workgroup resources or groupware functions. Furthermore, the proxy could examine retrieved documents for viruses or screen them in other ways before allowing them through the firewall. Likewise, if connections to the proxy are made via slow links, content distillation can be performed at this point. Again, the capabilities of the web are clearly increased by introducing

programmability into the proxy.

Servers are simply a special case of intermediaries. A server is responsible for producing the document that satisfies a request. Therefore, an intermediary that produces a document in response to a request is acting as a server. Because this is just one of the possible functions of an intermediary, an intermediary is more powerful than a server.

By pushing on the common browser-proxy-server model, we have uncovered programmable intermediaries at the client, proxy and server. The intermediary view is a generalization of the current web model that increases the web's flexibility and power.

3. Architecture

Intermediaries promise a more powerful and flexible mechanism for programming web applications. Yet many of the details of intermediaries are quite complex. They must handle protocol details, rare error conditions, common perversions of specifications and myriad other special cases. A number of monolithic intermediary-based applications have been built by different groups, but the potential will not be realized unless they are made easier to program. To solve this problem, we have designed an intermediary architecture that takes care of the details of intermediary plumbing, allowing the application programmer to concentrate on the data manipulations necessary for particular applications.

Our architecture for intermediaries is known as Web Browser Intelligence (WBI, pronounced "Webby"; [1]). WBI is a programmable proxy server that was designed for easy development and deployment of intermediary applications. Our goal was to produce a programming platform that can be used to implement all sorts of intermediaries, from simple server functions to complex image and page editing and distributed applications. The WBI architecture has been used by more than 20 programmers for numerous projects including intelligent agents, web mapping, password management, document format conversion, collaborative filtering, and knowledge management.

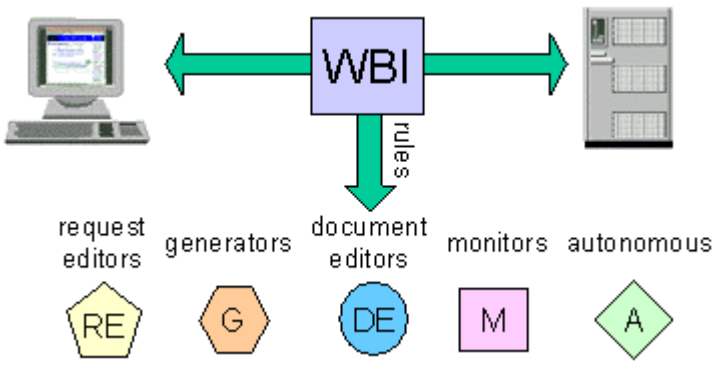

Figure 3. Intermediary functions are constructed from 5 building blocks. Each is connected to WBI through a rule that specifies what type of transactions it should be involved in. For example, a monitor could have a rule that specifies that it only wants to monitor JPEG images delivered through the HTTP protocol.

3.1 Building Blocks

In WBI, intermediary applications are constructed from five basic building blocks: request editors, generators, document editors, monitors, and autonomous functions (see Figure 3). We refer to these collectively as *MEGs*,

for Monitor/Editor/Generator. Table 1 describes these different elements and their functions. Monitors observe transactions without affecting them. Editors modify outgoing requests or incoming documents. Generators produce documents in response to requests. Autonomous functions run independently of any transaction and perform background tasks.

MEG Type	Input	Output	Action	Examples
Request Editor	Request	Request	Modify request	Redirect the request to a new URL, modify header information, insert form information, add/remove cookies
Generator	Request	Document	Produce document that satisfies request, or reject request	Read the document from a locally-stored file, forward the request to another intermediary or server, dynamically compute the document (like CGI), compose the document from a database, produce a control page such as "document moved" or "invalid authorization"
Document Editor	Request and Document	Document	Modify document (which could be text, HTML, image, applet, etc.)	Add annotations, highlight links, add toolbars, translate document format (e.g. from Rich Text Format to HTML), change form information, add scripts
Monitor	Request and Document	None	Receive request and resulting document, and perform computation	Gather usage statistics, record user trail, store documents in a cache, record filled-out forms
Autonomous	None	None	Perform background computation that is not affiliated with a particular transaction	Calculate user browsing statistics, search the web for new documents, crawl a server for broken links, cleanup unneeded cache files

Table 1. The basic MEG (Monitor/Editor/Generator) building blocks used to build intermediary applications.

3.2 WBI Operation

WBI dynamically constructs a data path through the various MEGs for each transaction. To configure the route for a particular request, WBI has a rule and a priority number associated with each MEG. The rule specifies a boolean condition that indicates whether the MEG should be involved in a transaction. The boolean condition may test any aspect of the request header or document header, including the URL, content-type, client address, server name, etc. Priority numbers are used to order the MEGs whose rules are satisfied by a given request.

When it receives a request, WBI follows these steps:

1. The original request is compared with the rules for all Request Editors. The Request Editors whose rule

conditions are satisfied by the request are allowed to edit the request in priority order.
2. The request that results from this Request Editor chain is compared with the rules for all Generators. The request is sent to the highest priority Generator whose rule is satisfied. If that Generator rejects the request, subsequent valid Generators are called in priority order until one produces a document.
3. The request and document are used to determine which Document Editors and Monitors should see the document on its way back to the original requester. The document is modified by each Document Editor whose rule conditions are satisfied in priority order. Monitors are also configured to monitor the document either (a) as it is produced by the generator, (b) as it is delivered from the intermediary, or (c) after a particular Document Editor.
4. Finally, the document is delivered to the requester, which may be the browser if this is the first intermediary in the chain.

An application is usually composed of a number of MEGs which operate in concert to produce a new function. Such a group of MEGs is a *WBI Plugin*. Plugins define the basic unit of granularity for installation, configuration, and enable/disable.

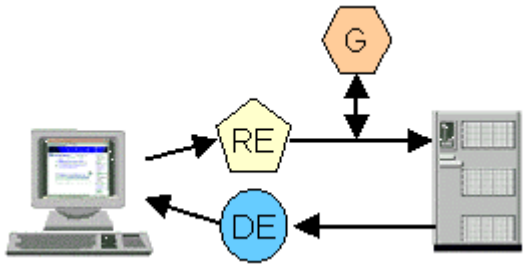

Figure 4. A simple cookie manager WBI Plugin built out of three MEGs. The Request Editor adds cookies from the cookie store (not shown) to the appropriate requests. The Document Editor removes cookies from the delivered documents and adds them to the cookie store. The Generator produces and interprets various HTML forms so the user can manage the cookie store and its policies.

3.3 Example WBI Plugins

3.3.1 Cookie Manager

Figure 4 illustrates a simple WBI Plugin for managing HTTP cookies with an intermediary. Current browsers allow the server to send a *cookie* with a document. The cookie is stored on the client machine and can be used to identify the user later. Some users do not want cookies to be stored, as cookies remove a certain degree of privacy. Some browsers allow the user to configure a cookie policy (e.g., always accept cookies, never accept cookies, or ask the user for each cookie). But these simple policies are often not sufficient, and it is often difficult to examine the cookies that have already been stored. With WBI, it is very easy to build a flexible cookie manager. A Document Editor modifies all documents that are sent to the browser. If it detects a cookie, it removes the cookie, only storing it if it fits a preselected policy. If the policy permits, a Request Editor adds cookies back into requests before they are sent to the server. A Generator produces and interprets some simple HTML forms that allow the user to set the cookie policy and to review and delete stored cookies.

The cookie manager is a simple example of a WBI Plugin built from three MEGs arranged to work in concert to form a single, useful function. It also illustrates some of the advantages of building functions with intermediaries rather than building functions directly into browsers:

1. the same cookies are available on multiple browsers

2. the same cookie manager program can be used on a workgroup server, allowing many users access to this function without requiring them to install it on their personal machines
3. new cookie manager plugins can be developed for special needs

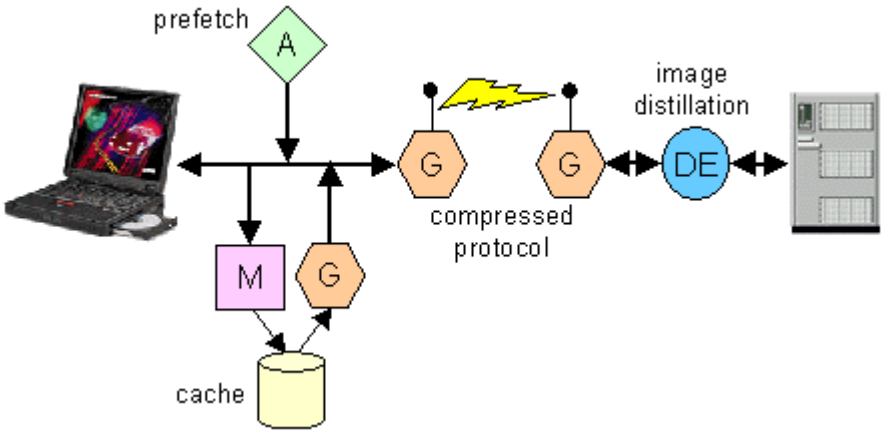

Figure 5. Several WBI Plugins that work together to improve a wireless web browser. A Monitor and Generator manage a local cache. An Autonomous function periodically prefetches important documents. A pair of Generators, one on the client and one at the workgroup server convert the standard HTTP protocol to a compressed version suitable for wireless transmission. A Document Editor distills documents to reduce image size and complexity.

3.3.2 Improving Wireless Performance

A more complex set of WBI Plugins is illustrated in Figure 5. The goal here is to make a wireless Internet connection more efficient. One set of client-side MEGs manages a cache that has a prefetch capability. A Monitor observes all web transactions and stores the results in a disk cache. An Autonomous function periodically requests commonly-needed documents to refresh the cache for instant access. A Generator intercepts all requests from the browser, examines the cache and, if possible, satisfies the requests. If the Generator cannot find the desired document in the cache, it rejects the request. The next Generator in the chain translates the HTTP protocol into a special, compressed version that optimizes the throughput of the wireless link, and sends the translated request across the link. The compressed request is received by a workgroup intermediary that contains a Generator to de-translate the compressed protocol back into HTTP and pass it to the requested server. Finally, a Document Editor distills what comes back from the server to remove unneeded color depth before sending the document to the client.

This application breaks down into three plugins: a client-side cache with prefetch, a compressed protocol translator, and an image distiller. Each is useful in its own right, but together form a powerful means for improving web browsing for the mobile user. Modularity makes intermediary programming productive and intermediary applications flexible.

4. WBI Implementation

Two equivalent versions of the WBI infrastructure have been built, one in C++ (both 32-bit Windows and OS/2) and the other in Java. Both allow WBI Plugins to be written in C, C++ or Java. WBI can serve as an intermediary for HTTP, HTTP-S, FTP and Gopher protocols. It can handle proxy- or server-type requests, and the rule system can differentiate between these to route requests appropriately. Therefore WBI can be accessed directly from a browser just like any server, or the browser can be configured to use WBI as a proxy.

4.1 Configurations

One typical WBI configuration is to install it on a client machine and configure the client browser to use "localhost" as its proxy. This *client-side* intermediary processes all requests from that browser, and can provide access to local information, personalization functions, or can perform other client-specific computation. WBI can be configured with any appropriate firewall information so that normal web requests are passed through a socks server or HTTP proxy.

The *workgroup* configuration is similar to the client-side case, except that several client machines are configured to use the same instance of WBI as their proxy. WBI can be running on one of the client machines or on any other networked machine. Some functions require that different users be identified, so WBI provides the capability of identifying individual users through the HTTP Proxy Authentication mechanism. Thus, functions within the workgroup intermediary can keep users separate to maintain individual histories or to provide custom configurations.

Finally, a configuration switch can set WBI to be a *server-side* intermediary. When this switch is set and a server name is specified, server-type requests run through the normal rule system and MEGs. But if no Generator accepts the request, it is forwarded to the specified server. In this way, WBI can appear to a browser as if it were the server, but it can provide additional functions. For example, WBI can translate documents that come from the server or it can provide search services for the server. This can all be done without configuring the destination server in any way: users simply access the server-side WBI instead of the server to get the enhanced functions.

4.2 Component Model

Recently, WBI's plugin definition has been adjusted to fit the Java Beans component model. In this model, WBI Plugins beans contain MEG beans. When WBI starts up, each registered Plugin is initialized. In their initialization code, the Plugins register with WBI their various MEG beans as *transaction listeners*. Each MEG bean contains properties that define the MEG's firing rule, priority, and other pertinent information. A standard library of beans has been developed for common functions, such as producing a static page of HTML, serving a file from a disk, adding annotations to the top of web pages, interpreting forms, and generating "document moved" responses.

The bean model enables programmers to easily develop systems with a visual programming environment. Because WBI beans are not graphical, the simplest bean builders do not provide an appropriate user interface. We have begun to develop a visual programming environment for WBI that allows the developer to build MEGs from other MEG beans and to incorporate them into Plugins. The environment also has facilities for visually tracing transactions through the MEGs and for viewing the results after each edit or generate step.

We have found the bean model to greatly increase programmer productivity as common beans are combined to form complex MEGs and Plugins. Most applications still require some code to be written, but often this code simply instantiates basic MEG beans, sets their properties appropriately and then forwards requests to them.

4.3 Performance

Intermediary performance is an important issue for practical implementations. First, it should be noted that the standard browser/firewall/server model is a special case of an intermediary model. In this case, the firewall is a simple intermediary that forwards requests to an external network and the server is a simple intermediary that delivers file-based or dynamic content. Therefore, it is clear that intermediaries do not fundamentally degrade web performance.

But what price is paid for a fully programmable intermediary such as WBI? Preliminary performance tests using

WebStone 2.0 show that the C++ implementation of a WBI-based server runs 1.2 times slower than Microsoft Internet Information Server on Windows NT. The Java version of the WBI-based server, using Sun JDK 1.1.3 with the JPP 1.1.3 just-in-time compiler, runs 2.5 times slower than the C++ version. This result shows that running requests through our native-code rule system and MEG dispatcher requires 20% processing overhead. The Java benchmark indicates the efficiency of current Java implementations. We believe that the extra CPU load is not unreasonable considering the advantages of the intermediary programming model.

Of course, adding intermediary functions beyond simple web serving will take more processing, but the bulk of this will be in the particular enhancement functions, with little additional system overhead. In more pragmatic terms, we have found that both the C++ and Java versions of WBI can add considerable functionality when run as a personal client-side intermediary without the user noticing performance degradation. In the workgroup and server configurations, more care must be taken to ensure that there is enough CPU power to handle the extra computations.

5. Discussion

5.1 Comparison with other web programming tools

WBI can be compared with many other programming paradigms for web-based applications, including CGI, NSAPI, ASP, OreO, and Servlets. Each has strengths and weaknesses, yet WBI is the first to fully implement the intermediary notion. Rather than compare WBI to each of these, we choose to consider JavaSoft's Servlets in more depth; the results are summarized in Table 2. (A comparison between WBI and OreO is given elsewhere [1]).

	WBI	Servlets
Language	Java, C, C++	Java
Location	client, server, workgroup	server
Operations	request editor, generator, document editor, monitor, autonomous	servlet
Granularity	MEG, Plugin	servlet
Rules	boolean expressions across request and document header fields	URL/content-type matching
Configuration	dynamic	static
Negotiation	priority system, Generators may reject requests	manually ordered
Code Location	local	local or network

Table 2. A comparison of programming with WBI and Servlets.

WBI is a multi-language system with the base code in both C++ and Java. WBI Plugins can be written in C, C++, or Java and these plugins can operate equivalently with either base. This cross-language compatibility is valuable but may not be maintained in future versions of WBI because of the unacceptable restrictions that result from programming to the lowest common denominator between C++ and Java. Servlets are fundamentally based on Java.

WBI's programming model places equivalent programmability at the client, server and workgroup server. Servlets, as the name implies are designed to be run on a server in the traditional model.

WBI has five basic building blocks for constructing applications, Servlets only has one. Though all of the functions can be performed with Servlets, they are not optimized. For example, WBI runs Monitors at low priority because they do not affect the user interface. To implement a Monitor with Servlets, the programmer would have to write an editing servlet that simply copied its input to its output (an unnecessary computational load) and then perform the monitoring function.

Servlets are configured one servlet at a time while WBI has a Plugin granularity that allows multiple MEGs to be configured at once. This plugin level of granularity is more useful for users and system administrators.

WBI's rule system for dynamically determining the data path through the various MEGs allows arbitrarily complex boolean expressions. Servlets are limited to simple URL and content-type matching. With WBI, rules can be dynamically altered to add or delete MEGs, or to change their firing condition, all under program control. WBI's priority system allows dynamic negotiation of the data path through the MEGs. In addition, Generators may selectively reject requests so that, though the rules might indicate that the Generator can handle the request, the Generator itself can reject it later. This feature is crucial for implementing important functions such as a cache. The flow of transactions through Servlets is statically configured at server startup or is manually adjusted by a system administrator.

Servlets do have a couple of features that are missing from WBI. Specifically, the configuration can load Servlets from remote servers rather than requiring them to be manually loaded on the same machine. In addition, a special <SERVLET> tag allows Servlets to insert dynamic content into the document. These functions can be added to WBI without difficulty.

We have found the WBI architecture to be powerfully productive. Though similar to several other systems, the full-featured intermediary paradigm is unique and gives the programmer unparalleled flexibility.

5.2 Metadata

The role of intermediaries grows more important as the use of machine-readable metadata increases. At present, the web is largely designed to be human-readable. As a result, it is often difficult for a computer to parse the contents of a page---even the basic word breaks and word order is often obscured in the interest of producing a particular visual presentation. It is even more difficult for a computer to determine important facts from a web page. For example, what is the item, price and merchant on an advertisement page? Metadata standards and incentives for their use will lead to an information environment where computers can handle information intelligently rather than simply transmitting it. The establishment of the PICS standard and associated languages for communicating page content is a good first step. When such metadata is embraced by the community, intermediaries will become vastly more useful. For example, they will be able to compile information from disparate sources, produce customized views and reports, and evaluate timeliness and quality of information.

5.3 Extending WBI

WBI implements the intermediary concept for the basic web protocols, principally HTTP and HTML. But the utility of intermediaries is not limited to these particular cases. We are exploring the extension of WBI to handle other protocols such as news and email. Example applications include automatic routing of email questions to known experts, mail summarization, and virtual newsgroups compiled from across the entire news feed. We are even looking at intermediaries for non-digital information streams such as telephone (e.g., Wildfire).

As intermediaries handle more information streams, it becomes advantageous for them to be able to share information. For example, a mail intermediary that caches received mail can make that mail available through

HTML web pages via an HTTP intermediary. To implement this, the mail intermediary and web intermediary need to share the mail contents. In another project, User-Centered Push [3], our group has implemented a blackboard architecture for sharing such information. Essentially the blackboard becomes a clearinghouse for user modeling and user interest information, which is made available to all of the intermediaries.

5.4. Summary

Intermediaries extend the web model to include content computation at client and workgroup proxy as well as at the server. This programming model provides a new degree of flexibility and allows web applications to do things that they could not do before. The WBI architecture is a productive infrastructure for programming web applications using intermediaries. Intermediaries provides a new place for applications to manipulate web data.

6. References

1. Barrett, R., Maglio, P. P., & Kellem, D. C. (1997) How to personalize the web. In *Proceedings of the conference on human factors in computing systems (CHI '97)*. New York: ACM Press.
2. Fox, A., & Brewer, E. A. (1996). Reducing WWW latency and bandwidth requirements by real-time distillation. In *Proceedings of the fifth international World-Wide Web conference.* (Also available as Reducing WWW Latency...).
3. Underwood, G., Maglio, P. P., & Barrett, R. (1997). User-centered push for timely information delivery. Manuscript submitted for publication.

7. URLs

Common Gateway Interface
http://hoohoo.ncsa.uiuc.edu/cgi/
Developing Web Applications for IIS
http://www.microsoft.com/iis/usingiis/developing/#ASP
Hypertext Transfer Protocol Overview
http://www.w3.org/Protocols/
Hypertext Transfer Protocol -- HTTP/1.1
http://www.w3.org/Protocols/rfc2068/rfc2068
IBM ARC - USER: WBI - Web Browser Intelligence
http://wwwcssrv.almaden.ibm.com/wbi/
Internet Information Server
http://www.microsoft.com/products/prodref/81_ov.htm
Java Beans - The Only Component Architecture for Java
http://www.javasoft.com/beans/
Java Servlets
http://www.javasoft.com/products/java-server/servlets/index.html
PalmPilot: The Connected Organizer
http://www.usr.com/palmpilot/
Platform for Internet Content Selection
http://www.w3.org/PICS/
Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation
http://www5conf.inria.fr/fich_html/papers/P48/Overview.html
Programming with the NSAPI
http://search.netscape.com/misc/developer/conference/proceedings/s5/index.html
RFC 2109 - HTTP State Management Mechanism
http://www.cis.ohio-state.edu/htbin/rfc/rfc2109.html
Transducers and Associates: Circumventing Limitations of the World Wide Web
http://www.camb.opengroup.org/www/waiba/papers/etacom/etacom.html

WebStone: Performance Benchmarking
http://www.sgi.com/Products/WebFORCE/WebStone/
Wildfire
http://www.virtuosity.com/html/wildfireframes.html