Dynamic Web Stream Customizers

Jesse Steinberg and Joseph Pasquale
Department of Computer Science and Engineering
University Of California, San Diego

Abstract— **We present an architecture for Web Stream Customization (WSC) which allows users to customize their view of the Web for optimal interaction and system operation when using non-traditional client machines such as wireless palmtops. Our Web stream customizers are dynamically deployable and can be strategically located to achieve improvements in performance, reliability, or security. Customizers provide two points of control in the communication path between client and server, supporting adaptive system-based and content-based customization. Our architecture exploits HTTP's proxy capabilities, allowing Customizers to be seamlessly integrated with the basic Web transaction model. We describe the WSC architecture and its implementation, and illustrate its use with three non-trivial, adaptive Customizer applications that we have built. Our performance evaluation of the system shows that the overhead introduced is small and tolerable, and is outweighed by the benefits that Customizers provide.**

Keywords*:* HTTP, Middleware, Proxy, Wireless, Mobile Code

I. INTRODUCTION

To a large degree, the Web has developed on an Internet infrastructure consisting of increasingly higher-bandwidth, lower-error-rate network links. The protocols and models of user-interaction, which work well under such conditions, are often not adequate for many of today's new wireless Internet personal devices. We must now take greater account of problems such as intermittent and lower-bandwidth connectivity, smaller displays, stylus-based input mechanisms, power restrictions, and limited memory.

In addition to the changing landscape of user devices, the increased popularity of using the Internet for a variety of tasks introduces challenges regarding privacy and convenience. Although sites that require credit card numbers typically use secure connections, there are still many insecure sites that require registration that includes personal information such as a mailing address, phone number, or e-mail address. And regarding lack of convenience, Web users often find themselves deluged with unwanted data such as advertisements, which slow down the transfer of legitimate information.

Users would indeed like to be able to *customize* their view of the Web, including removing data they are not interested in downloading, filtering images to smaller representations, and displaying Web pages in an easy-to-surf format. To limit bandwidth usage over a wireless link (which generally has lower bandwidth and reliability, and less security due to the ease of eavesdropping, than wired portions of the Internet), content should be compressed and possibly encrypted at some point before the wireless link and then decompressed and

decrypted at the client. Provisions should also be made for helping users deal with disruption of important transactions and masking short-term intermittent disconnections.

Furthermore, these types of customization should be able to dynamically adapt to changes in system conditions or to user behavior. For example, if network throughput is relatively high, compression may not be beneficial if it takes too much time for a low-powered device to perform the decompression. However, as available throughput decreases, the performance gain of compressing data prior to transferring it over a slow link begins to outweigh this drawback.

There are many types of Web customization possible, and it is useful to broadly characterize them. We define two such categories, *system-based* and *content-based,* which cover a wide range of possible customizations that lead to improvements in performance, reliability and security of the Web. By system-based, we mean customizations that mostly rely on (and exploit) characteristics of the underlying system, including hardware or software aspects of the client, the server, and the intervening network connections. This includes static resources and their characteristics, such as the client's display size and memory, and dynamic resources such as available network throughput, and connectivity. By content-based, we mean customizations based on the actual content of what is being communicated. Examples include advertisement filtering and selective encryption of embedded private information.

In this paper, we present a new system for Web customization based on the concept of Web Stream Customizers (WSC), or simply Customizers, and we illustrate their use via a set of applications that we have built and found useful. Customizers are distributed web customization software modules that are dynamically deployed and used by clients during a Web session (although servers and even third parties can deploy and use them). Customizers are seamlessly integrated with the basic Web transaction model, simplifying their programming and operation. This is because the WSC system exploits the Web's proxy capabilities, and makes use of standard code mobility mechanisms (with Java as the language of choice given its portability). Thus, importantly, Customizers will work with standard browsers and Web servers, without requiring any modifications to them.

A key feature of the WSC architecture is that it supports cooperative customization at two points along the path between client and server. Many types of customizations, both system-based and content-based, require such cooperation and distribution of functionality. For example, data compression (e.g., to reduce bandwidth requirements, and perhaps latency) requires that compressing be done before the data crosses any relatively low-bandwidth links, and that decompressing be done afterwards.

Another key point is that customizers provide client-specific customization of server content, effectively adapting the Web to new and different sorts of clients. This even includes the ability to dynamically and easily deploy client-specific (or application-specific) protocols, such as to deal with problematic network connections. The ability to deploy such protocols relies on the two-point distributed operation.

An alternative approach to adapting the Web is to introduce a new system of protocols and document types to address specific problems, as typified by the Wireless Access Protocol (WAP) for Internet access by wireless clients. The problem with such approaches is that they require changes to actual Web servers; those that do not will be effectively inaccessible. Considering the vast amount of legacy content in existence today, a more flexible and universal way of enhancing the Web experience when accessing arbitrary services, *working within existing Web and Internet structures*, is needed.

The more traditional approach to adaptability that does not depend on Web servers to cater to client-specific needs has been to use fixed proxy servers, which are indeed supported by the HTTP standard. Web proxies act as intermediaries between clients and servers by intercepting client requests and forwarding them to servers, eventually returning the server response to the client. All the popular Web browsers have a setting that allows the user to specify a proxy to be used for all requests. Proxies have been used for HTML filtering, user interface improvements especially for small screens, remote caching, and support for disconnected operation and user-selected background retrieval [4,8,5,6,13]. Although they are beneficial, traditional proxies are statically located and often too limited to provide the wide range of customization that can benefit users, such as those requiring cooperation at both ends of a communication link (or a string of them), and even moreso if they are to act on a per-request or per-server basis.

Other approaches include combining proxies with mobile code, or using generalized mobile code systems to customize the Web [25,18,21,11,9,17,20]. Our work differs in a number of ways. First, we have focused on a customization system *designed specifically for the Web*, allowing us to make a number of simplifying assumptions regarding the programming model, the user model, and the system design and implementation. Second, we use a very restricted form of mobile code, rather than providing a generalized mobile code solution which, while more powerful, is less practical and is more complex in terms of usability and security.

Other unique features of our system include the dynamic selection of one of multiple, simultaneously active, Customizers, based on which server is providing the content. As mentioned above, our Customizers have two components; one is statically located either on or close to the client, and the other is dynamically located at a convenient point between the client and server (typically near the server). Finally, we use a simple, callback-based programming model, and support user-controlled deployment, including the strategic placement of the remote component, through a simple Web interface. We will elaborate on all these issues in this paper.

The remainder of this paper is organized as follows: In Section II we provide an overview of the WSC architecture. The implementation details are described in Section III. In Section IV we illustrate some example Customizer Applications. In Section V we analyze the performance of Customizers. Section VI details related work, and in Section VII we present our conclusions.

II. THE WSC ARCHITECTURE

There are three primary goals that drive our design of the WSC architecture:

- **Flexibility:** The location of where the remote customization is carried out, and the type of customization being done, can all be determined dynamically.
- **Ease of deployment:** The system does not require changes to the client's Web browser or any Web servers, and works within the current Web transaction model using HTTP and existing mobile code mechanisms.
- **User simplicity:** The interface allows the user to install, invoke and configure Customizers via a standard Web browser using the basic "Web surfing" model of interaction.

The WSC architecture enables Web customization without modifying the browser client or any Web servers by introducing Customizers that operate between them. When a client generates a Web request, that request is transparently routed to a specific Customizer, selected based on the URL of the request. The Customizer then has the opportunity to modify the request if it so chooses, and then forwards it to the Web server as indicated by the URL. The response from the Web server is then routed back to the Customizer, which has the opportunity to modify it before passing it back to the client. As one can see, conceptually, the idea is very simple. The client sees a Customizer as a proxy, which is then viewed by the Web server as a client. While this simplified view is shared by other approaches to customization, our system deviates in the details that now follow.

Actually, a Customizer is comprised of two components: a *local component* (LC) and a *remote component* (RC). The LC runs on a *Local Customizer Server* (LC-Server), and the RC runs on a *Remote Customizer Server* (RC-Server), as shown in Figure 1. Thus, when a Customizer is being used, the request passes from the client to the LC, then to the RC, and then to the server (and vice-versa for responses in the opposite direction, from server to RC to LC to client). Examples of how an LC and RC cooperate will be given throughout this paper.

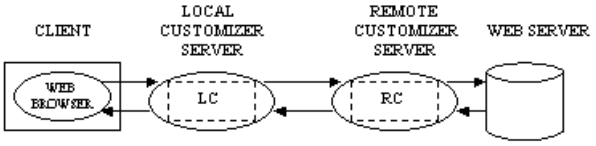

Fig. 1. Local and Remote Components of a Customizer Running on Local and RC-Servers

Why separate a Customizer into two components? The LC and RC have distinct roles. The LC acts primarily as an extension of the browser (given that the browser code itself cannot be modified). The LC runs on an LC-Server, which tends to be located on or near the client Web device. Given its close coupling with the client, the LC is generally responsible for tasks that require knowledge of resource availability and system conditions at or near the client, which may then be communicated to the RC (e.g., to improve performance, such as relaying local system or network performance status). In addition, the LC will also reverse data transformations done by

the RC, such as compression/decompression or encryption/decryption.

The RC generally performs location-dependent tasks that benefit from being near the server (or simply away from the client), such as compressing response data from a server before it is transmitted over a low-bandwidth link on the communication path to the client. The RC runs on an RC-Server, which tends to be located near, or even on, a Web server of particular interest.

There will generally be many Customizers, each one being a separate (LC, RC) pair, simultaneously active on behalf of a single client. All of the LCs (for that client) will run on a single LC-Server; since all the LCs originated from that client, they will all run on that LC-Server. This is in contrast to the RCs, which will be generally running on different RC-Servers, as shown in Figure 2.

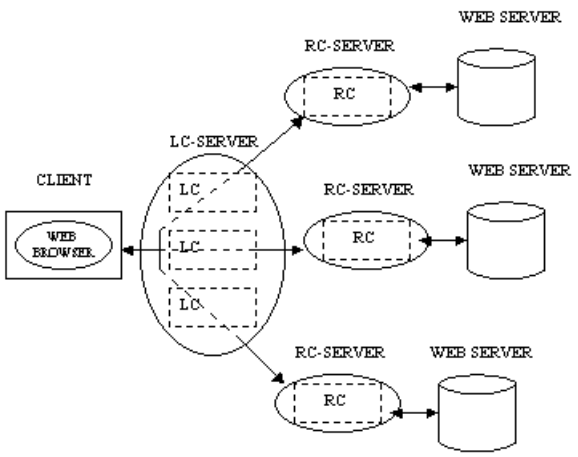

Fig. 2. An Example of Using Multiple Customizers

Let us elaborate on the services provided by LC-Servers and RC-Servers. The LC-Server effectively extends the client machine by hosting LCs for the browser running on that machine. It appears to the browser as a Web proxy server. The browser needs only to have its proxy server settings configured to point to the LC-Server.

For a particular session of Web browsing, the same LC-Server is always used. This is to avoid having to frequently modify the browser's proxy settings in order for it to interface with the many Customizers that may be active. While the LC-Server location is static (per session), flexibility is achieved by allowing multiple RC-Servers to be used during a single session. A common scenario would be to have a number of Customizers active at any time, with many RCs running on various RC-Servers, each with a corresponding LC running on the LC-Server associated with the client, as shown in Figure 2.

Since there is a single LC-Server per session, in order for the location of some forms of customization to be chosen dynamically, the LC-Server must be able to dynamically route Web requests to different RCs. The LC-Server chooses an RC on a particular RC-Server based upon the URL of the request. This will be explained in Section III.

The LC-Server will often be running on the client machine, co-located with the browser. Running the LC-Server on some other machine is useful if the client machine is not powerful enough or is incapable of running the LC-Server process, as may be the case with a PDA limited to running pre-installed applications. (Note that even in the case of a limited PDA, we expect the PDA to be able to at least run a browser capable of being configured to communicate via a proxy. This is a basic requirement of our system. It is worth noting that our current experience with PDAs equipped with browsers, such as the

Compaq iPaq, HP Jornada, and others, is that they all have this basic capability.)

The role of the RC-Server is to load and run RCs that benefit from executing at a remote location. An RC-Server can only be used in conjunction with an LC-Server. As mentioned above, unlike the LC-Server, a client may be actively using a number of RC-Servers for handling communication with different Web servers. Thus, for a single session of Web browsing, many different RC-Servers may be used for varying amounts of time. Each of these RC-Servers may have any number of RCs loaded and running at any time.

Use of both an LC and RC provides for two points of control, allowing for flexible distribution of functionality as determined by the programmer. More generally, the local/remote combination provides the capability of supporting custom protocols that optimize transfer over the last one or more hops, data transformations that need to be reversed such as compression and encryption, and any other form of customization that requires both local and remote functionality, examples of which shall be described in Section IV.

To summarize these concepts, Figure 3 shows a simple and common scenario of using a Customizer for data compression. The figure depicts three machines: a wireless notebook client, a base station, and a Web server. The user's client machine is running a Web browser, and the base station is running an RC-Server that can control and service one or more RCs on behalf of the user (and other users). The LC-Server is running on the client. A Compression RC is running on the RC-Server to compress responses from Web servers before they are sent across the wireless link. There is a Decompression LC on the LC-Server, which cooperates with the Compression RC by decompressing the compressed Web responses to their original format so that they can be displayed by the browser. The Compression RC and the Decompression LC together make up the single Customizer. The positioning of the Customizer components in this example allows data to be compressed before crossing the low-bandwidth wireless link, reducing download times.

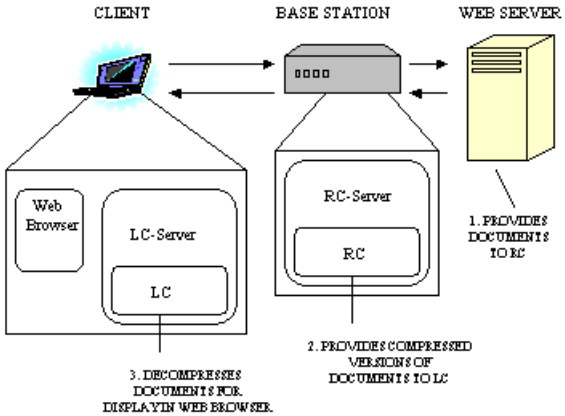

Fig. 3. A Compression Customizer

III. The WSC Implementation

We chose to implement the WSC architecture in Java because of the widespread availability (actual or potential) of Java Virtual Machines, providing a ubiquitous platform to support Customizers, and because it supports code mobility for the dynamic loading of LCs from an RC-Server to an LC-Server (this is an important feature of our implementation which will be described below). The Customizer Servers are implemented as Java applications, and the LC and RC are

made up of one or more Java class files packaged into a Java Archive File (jar). In addition to the architectural goals of flexibility, ease of deployment, and user simplicity, ease of programming Customizers was another goal of the implementation.

A. *Customizers and the Server Execution Environment*

Both the LC and RC consist of a Java class that implements a new interface called the Customizer interface, along with any other Java classes they may use, all packaged into a jar file. The fundamental method defined by the Customizer interface is *HandleRequest.* It is via this method that the Customizer components actually have the opportunity to view and customize the Web object that is being requested.

When a Customizer component is loaded, it actually runs as part of a LC-Server or RC-Server, which we will more generally refer to as a C-Server. It is the C-Server that invokes a component's *HandleRequest* method to act on a Web request. We chose this "callback" style of invocation for numerous reasons, including security, ease of programmability of Customizers, and ease of deployment and integration with the Web. Regarding security, we rely on Java language mechanisms, including support for a security manager object that provides coarse-grained control over what resources objects can access. In the WSC architecture, we rely on the security manager to prevent the ability of a Customizer component to access resources such as network and disk I/O. Only the C-Server is allowed to do network or disk I/O.

Hence, to allow for HTTP customization under these strict restrictions, we adopted the callback style of invocation for the Customizer component by a C-Server. The callback model is widely used in Java programming for the Web. For example, in the Applet model, there are callback methods such as start() and stop() that are called by the runtime system when the Applet is started and stopped based on the user entering and leaving the Web page. The Java event model for handling user interface events uses listener objects to listen for events by the user, and methods in the listener object are called when an event occurs. Callbacks are also used in the Java Servlet programming model [19].

Using the callback model has the effect that Customizer components do not need to participate directly in network communication. Instead, it is the C-Servers that handle *all* of the communication, and pass Web request and response data buffers as parameters to a callback function implemented by the Customizer components.

B. *The LC-Server*

The LC-Server effectively links the Web browser to one or more Customizers, and handles the loading and configuration of Customizers through a Web interface. Recall that all requests made by the browser are received and handled by the LC-Server, which is viewed as a proxy by the browser. The LC-Server uses this mechanism to intercept the browser's requests for Web objects, and then to provide the requests to Customizers (by first handing it to the LC portion, and then

forwarding to the RC-Server hosting the RC portion). Although many Customizers can be active simultaneously with the remote components at different locations, the LC-Server is fixed, so the browser only needs to be configured once. This also means that the browser's settings do not need to change if there are changes to the configuration of particular Customizers being used.

In Section II we described the flow of information when just one Customizer is active. When multiple Customizers are active, the LC-Server must have a means of determining which Customizers should receive which browser requests, and hence, which RC-Servers will be involved in which requests. Consequently, each Customizer has associated with it a *Domain of Applicability* (DA), which specifies a set of URLs or host domains (i.e., Web servers), and this is stored at the LC-Server. When the LC-Server receives an HTTP request from the browser, it can determine if a particular Customizer should handle the request by checking whether the URL associated with the HTTP request is within that Customizer's DA; if so, that Customizer is used, and the request is first given to its LC portion, and then sent to the RC-Server hosting the corresponding RC portion, as shown in Figure 4. If a URL is common to the DAs of multiple Customizers, the current policy is to choose the Customizer that was loaded first. If no DA contains that URL, the LC-Server sends the request directly to the Web Server specified by the URL, bypassing all Customizers.

In addition to helping the LC-Server select a Customizer, the DA also helps the LC-Server protect a client's privacy interests. For example, a client may only want a particular Customizer to know about certain requests. By matching the DA to the client's requirements, this privacy can be ensured by rejecting a new Customizer that specifies a conflicting domain. Furthermore, there is a provision for allowing an LC-Server to impose a sub domain restriction on the Customizer if the URLs that the client is willing to show to the Customizer form a subset within the Customizer's domain. For example, a user may not want a Customizer to see all of its shopping-related Web requests, as it might use those for advertising purposes. The user could provide a list of their favorite shopping sites to the LC-Server, with instructions not to allow any Customizers to handle requests to these sites.

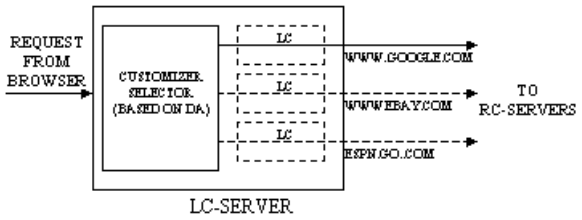

Fig. 4. Selecting A Customizer Based on the DA.

C. *Installing Customizers*

To make using Customizers as simple as possible and encourage their deployment, we have integrated Customizer installation and invocation into the already familiar Web surfing model. In other words, Customizers can be installed and invoked simply by the user clicking on hyperlinks during normal browser use.

To illustrate this, suppose a Web Server wishes to make available to clients a number of Customizers (which were specially programmed for this Web Server's content, providing highly content-specific customizations), with the remote components running on a nearby RC-Server. As shown in Figure 5a, the Web Server has a page (in standard HTML) that contains a list of Customizers. Each listed Customizer is a hyperlink pointing to a special file called a Customizer Metafile (CMF), which has a ".cmf" extension. This file provides the following information about a Customizer that is used by the LC-Server to load and invoke a Customizer:

- The hostname of the machine running the RC-Server that will run the RC of the Customizer
- The jar file containing the Java classes implementing the Customizer's components
- The name of the RC main class so that it can be loaded from the jar file
- The name of the LC main class so that it can be sent to the LC-Server
- Initial configuration parameters for the LC and RC;
- The DA (Domain of Applicability)
- Optionally, a URL for the Customizer's configuration page (described below).

Figure 5b shows the process of loading a Customizer by clicking on a hyperlink to a CMF. In Section II we described how the Web browser is set up to use an LC-Server as its proxy, and how this allows all Web requests made by the browser to go through the LC-Server. If a user clicks on one of the hyperlinks pointing to a CMF, the LC-Server intercepts this request, and retrieves the CMF from the Web server. Once the LC-Server has received the CMF, it can download the LC, associate the DA specified in the CMF with the Customizer, and send a message to the browser to inform the user that the Customizer was loaded. Future Web requests that are in the Customizer's DA will be sent to the Customizer by the LC-Server, as was shown in Figure 4.

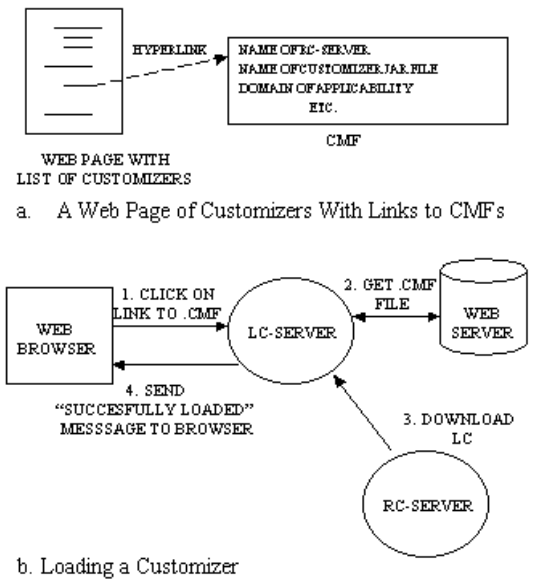

Fig. 5. Web-based Customizer Loading.

Note that the motivation for the dynamic downloading of LCs is that resource-limited, mobile clients can easily use them on the fly. Prior knowledge of the client's location is not required, and a client is does not need to store LCs that are not being used. The dynamic loading of the LC is similar to the popular Java Applet model of mobile code. The motivation for limiting our design to this basic model is to avoid introducing the additional system complexity and security

liabilities characteristic of more general mobile code mechanisms [7,24].

At any time, the client can go to a special Customizer control page made available by the LC-Server. This page has two major purposes. It can be used to directly control the use of Customizers, e.g., to turn Customizers on and off, or to unload them. In addition, the Customizer control page contains one link for each Customizer which, when clicked, will retrieve the *configuration page* for that Customizer. This page may be a static page simply provided by the Customizer (and will probably already be cached at the LC-Server), or the Customizer may actually generate it dynamically (since it can customize the request for the configuration page). A Customizer's configuration page allows the user to directly control parameters that affect the functionality of that Customizer. For example, an Image Filter Customizer could provide a configuration page with sliders that allows the user to control the extent of both reduction of image resolution and reduction of color-depth.

D. WSC Communication

Figure 6 shows how Web requests and responses are communicated between the client and server via the LC-Server and RC-Server for a typical Web request initiated by the Web browser when a Customizer is active. There are three TCP connections involved in a single Web transaction when Customizers are being used.

- *Connection 1* is between the Web browser and the LC-Server.
- *Connection 2* is between the LC-Server and the RC-Server.
- *Connection 3* is between the RC-Server and the Web server.

Each connection is used in the two phases of communication: the request phase where the request from the browser is received, and the response phase where the response from the Web server is sent back to the host that originated the request. Recall that Customizer components have the opportunity to participate in the interaction at both the LC-Server and the RC-Server.

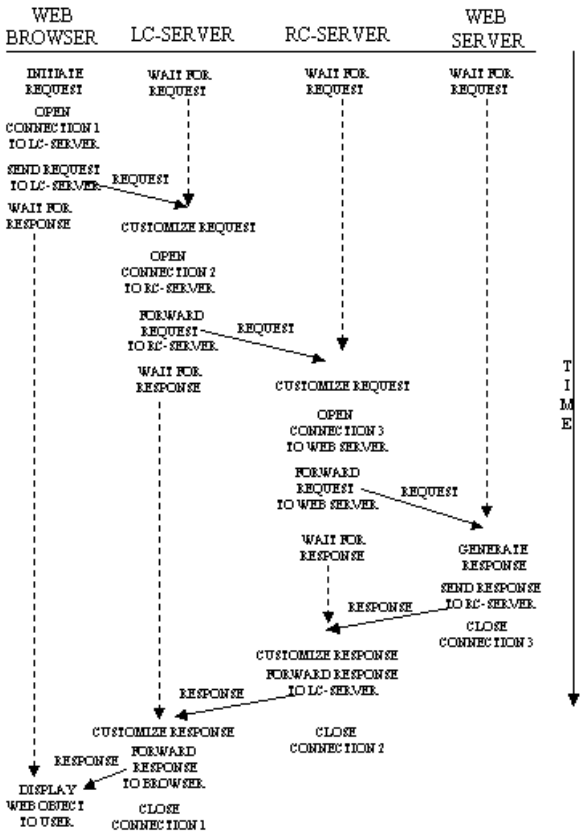

Fig. 6. WSC Communication.

The Web browser opens *Connection 1* automatically to handle a Web request, once it has been set to use the LC-Server as its proxy. The browser generates an HTTP request and sends it to the LC-Server. The browser then waits for a response in the form of the requested Web object from the LC-Server (just as it would had it sent the request directly to the Web Server). The LC-Server identifies the target LC and provides it the opportunity to modify the request. The LC-Server then opens *Connection 2* to the RC-Server. It forwards the browser's (possibly modified) request to the RC-Server and then waits for a response back from the RC-Server. The RC-Server works like the LC-Server: after receiving the request, it identifies the target RC and provides it the opportunity to modify the request. Then the RC-Server opens *Connection 3* to the Web server specified in the request URL. It forwards the request to the Web server and then awaits a response. The Web server receives the request, generates a response, and passes it back to the RC-Server. The Web Server then closes *Connection 3*.

Things work in similar fashion in the return direction. At the RC-Server, the RC is given an opportunity to customize the Web Server's response, and similarly for the LC at the LC-Server. Finally, the LC-Server returns the customized Web object to the Web browser. The browser can then display the response to the user. Note that we expect most of the action to take place in the return direction, as the response, i.e., the object being returned, is generally what is of interest for customization.

The details of callback-based invocation of Customizer components are shown in Figure 7. After the browser request gets forwarded to the LC-Server, the LC-Server checks the URL against its list of DA's for Customizers it has loaded. If the URL matches a DA, the LC-Server buffers the request and calls the *HandleRequest* method of the LC for that DA, with the request buffer as a parameter. By default, for security reasons, the LC cannot modify the request (which, as discussed below, is not the case for the response), although it can see the request and make decisions based on its content**.** However, a trusted LC can be given privileges to modify requests. This model allows the LC-Server, which is part of the standardized Customizer system code rather than the LC (which includes arbitrary code as written by Customizer programmers), to handle all explicit HTTP communication on behalf of the user. To send the request to the RC-Server, the LC calls a method provided to it by the LC-Server. This allows the LC-Server to handle all network I/O. The LC-Server then forwards the request to the RC-Server. By giving the LC the responsibility of calling *or not calling* the method, it has the power to decide not to forward the request to the RC-Server if it can handle it itself, e.g., as would be the case for a cache. (This is to be contrasted to the less optimal alternative control structure whereby the LC simply returns control to the LC-Server, which then automatically forwards it to the RC-Server.) The return value of this method is a buffer containing the response that comes back from the Web server via the RC-Server.

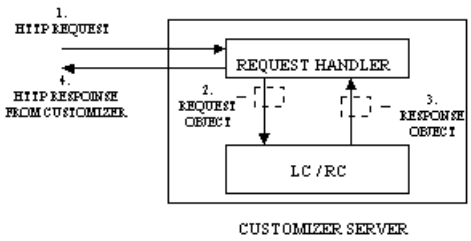

Fig. 7. The Callback Model

When the RC-Server receives the request from the LC-Server, it handles the request the same way as the LC-Server did. The request is buffered, and the *handleRequest* method of the RC is called. Again, only a trusted RC can actually modify the request, and the RC may forward the requests to the Web Server by calling method provided by the RC-Server or generate a response on its own without calling the method. The return value of this method is a buffer containing the customized response from the Web server, or a response generated by the RC.

To handle the response, just like for the request, the RC-Server actually handles the network I/O, and response buffers are passed to the RC. The RC-Server provides a service to the RC for returning the ultimate response to the LC-Server. Unlike in the forward path, by default *the RC may modify the response before it is returned to the LC-Server*. Once the LC-Server receives the response, it is returned to the LC, which also has the opportunity to customize it before it is sent to the Web browser.

Importantly, this basic customization mechanism uses only HTTP for communication, and is transparent to the browser. Privileged Customizers can use their own protocols to customize the request and response freely, while taking advantage of the services provided by the C-Servers to get the response from the Web server and pass the requests and responses along. Once a Customizer is installed, there is no special or additional interaction required by the user, as normal use of the Web browser will cause the Customizer to be invoked.

IV. Applications

We now present examples of three types of applications with which we have been experimenting: adaptive compression, transaction reliability, and privacy.

A. *Adaptive Compression*

Two examples of adaptive compression Customizers are a General Compressor and an Image Filter. The General Compressor is an extension of the Compression Customizer example described in Section II. It performs lossless compression on types of content that compress well, in order to reduce the amount of data transferred between the RC-Server and the LC-Server. This is beneficial when the RC-Server has a high-bandwidth, reliable connection to the Web Server, the LC-Server is on the client, and the client is connected via a a link characterized by low-bandwidth or low-reliability, as is the case for many types of wireless links.

This is a content-based form of customization because the RC will perform lossless compression only on content types amenable to compression such as text documents including HTML, plain text, postscript, and scripts (such as Javascript).

The RC supports multiple levels of compression so that the compression/decompression processing time and the reduction in network transfer time can be balanced. The LC serves two functions. First, it decompresses anything that has been compressed by the RC. Secondly, it measures response times so that it can it can tell the RC what level of compression to use, if any. Since the LC may be running on a low-powered client, decompression may be a performance bottleneck if the network throughput is relatively high. In this case, too much compression will be detrimental to overall performance. Hence, by keeping track of the changes in network performance, the LC can adapt the compression to the current conditions.

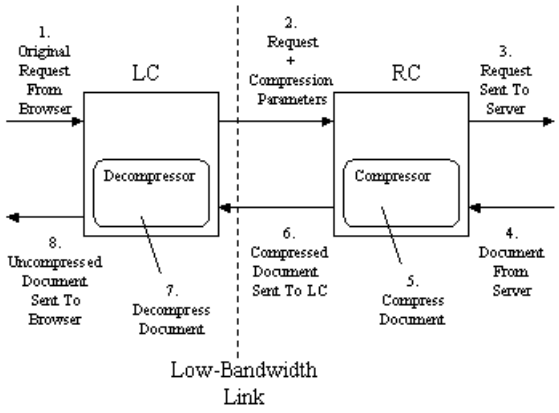

Fig. 8. The Compression Customizer

In our current implementation, the LC communicates customization parameters to the RC by adding unique HTTP headers to requests. This prevents any changes being made to the HTTP communication model supported by the WSC architecture. For example, when sending a document request to the RC, a header is added that specifies the desired compression level. All Customizer-specific communication described in this and other examples follow this procedure. Figure 8 shows the functioning of the General Compressor. For the sake of simplicity, C-Servers are not shown and communication is depicted logically between the LC and RC.

The Image Filter does adaptive lossy compression to improve performance by reducing image data. It is especially useful for wireless clients with low-bandwidth connections and small displays that cannot display large images with many colors. In this case, the RC should be running at a host with a reliable Internet connection that has sufficiently high bandwidth.

The RC handles the actual filtering. It provides three functions, all of which reduce the amount of data to be transferred from the RC to the LC:

- Scaling down the image size, based on a parameter specifying the maximum number of pixels (aspect ratio is maintained)
- Reducing the image color-depth by turning a color image into a grayscale image
- Converting images into formats that yield higher compression ratios.

The Image Filter is adaptive, much like the General Compressor, and adapts according to both network performance and user behavior. The role of LC is to provide information to the RC for adaptation. For each HTTP request

for an image, the LC can set the maximum number of pixels in the image if scaling is desired, whether the image should be in color or grayscale, and whether or not uncompressed or poorly compressed images should be converted into another format. Note that both conversion and filtering can be performed on the same image. The LC changes these settings based on user behavior and measurements of response times, in order to achieve the best performance under changing conditions. For example, if the user is accessing many web sites in parallel, each with many images, or if response times are currently very slow, the LC can reduce image size and color depth parameters sent to the RC so that each image uses less data. When the response times speed up, or the frequency of downloaded images reduces, the LC can recommend that filtering be turned off entirely.

The Image Filter RC also has to modify HTML pages containing images. The reason is that HTML pages often contain dimensions for inline images, which tell the browser how to layout the page before it receives the actual images. When images are scaled to smaller dimensions than those specified in the HTML code, the browser will automatically scale the images to fit the specified dimensions. This will cause the images to be displayed with a ”fat-pixel” effect. Hence, the dimensions specified in the HTML pages, must be changed to match the dimensions of the scaled image, so that the scaled images are displayed properly. Note that this may change the layout of the page from its designer’s intentions.

B. Transaction Reliability

We are experimenting with two examples of Customizers which help users deal with unreliable connections, the Connection Smoother and the Transaction Recorder. The Connection Smoother masks short-duration connection failures from clients with unreliable Internet connections. During normal Web surfing, a browser may request a number of documents in parallel. For example, if the user opens a page with many inline images, a separate connection may be used to request each of the images. If connectivity is lost while these requests are pending, the browser will display broken placeholders for inline objects and error messages for main objects such as an HTML page. The user will then have to reload the page after connectivity is reestablished.

Customizers can be used to mask such failures. Consider a scenario where the browser and LC are co-located on the same client machine (as in Figure 3), and the RC is running on a host that has a reliable connection to the Web Server. The connection between the LC and RC may be tenuous, e.g., a wireless link. As part of its normal operation, the RC can temporarily store Web objects and have them ready for retransmission in case it fails to fully send them to the LC running on the client. When the client's connectivity fails, the connection between the LC-Server and the RC-Server is lost, but the LC still has an open connection to the Web browser since they are on the same host. The LC continuously retries sending the request to the RC in case connectivity is reestablished in a short time. If connectivity is reestablished in

short order, the object is successfully retrieved and the response is sent to the browser, and the user will notice only a slight delay in the retrieval of Web objects.

As an added measure, each retry request contains a storage flag that informs the RC to use the previously stored object if it has one. If the storage flag is absent, the RC will request a new copy of the Web object from the destination Web server. The use of this flag allows normal web-surfing semantics to be maintained. Thus, if the user intentionally requests a new version of an object after a failure, they will not receive the stored version since the storage flag will not be set.

If the browser's connection times out before connectivity is reestablished, the LC can later send a special request to the RC which informs it to clear its object storage, since the stale objects are no longer needed. Alternatively, the RC can be configured to hold objects in storage for a specific amount of time, if it is important that memory usage be kept to a minimum. Figure 9 shows how Web transactions are handled by the Connection Smoother.

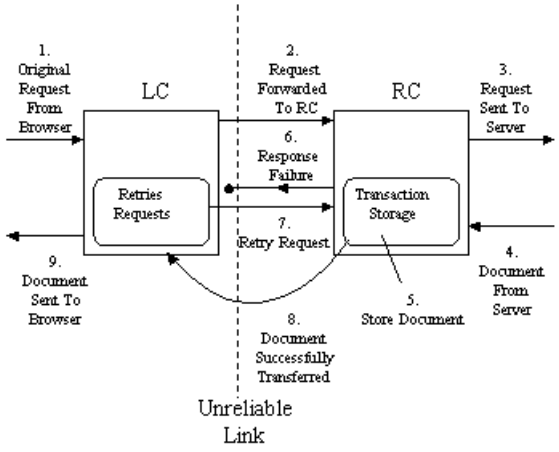

Fig. 9. The Connection Smoother

The Transaction Recorder stores recent Web transactions at the RC in case of failure. Unlike in the Connection Smoother, the LC does not automatically retry requests if connectivity is lost. Instead, when connectivity is reestablished, the user can bring up the Customizer's configuration page, which contains a list of all recorded transactions. This is useful for transactions, which should not be repeated such as transfer of money. As soon as connectivity is reestablished the user can easily discover the results of the transaction. If the results are not stored by the RC, then it did not receive the request, and hence the transaction was never executed. The configuration page allows the user to set the number of recent transactions to be stored at any time.

A variation of the Transaction Recorder is background retrieval of Web objects. The user controls the background retrieval by clicking on a link to a Web object, and then aborting the transaction by pressing the stop button on the Browser. Instead of the transaction being aborted entirely, the response is downloaded to the RC while the user is busy reading some other Web page. This requires cooperation by the LC, since only it knows when the browser closed a connection as a result of the user pushing the stop button (because the RC does not have a direct connection to the browser). If the user turns on background retrieval in the configuration page, then when they abort a transaction at the client, such as by pushing the stop button on the browser, the LC will not abort the transaction until after the request has been forwarded to the RC. Hence the RC will store the request. The user can then retrieve the object from the Transaction Recorder configuration page, which lists all stored objects.

This functionality is useful when dealing with slow servers. For clients with adequate memory, the objects can be stored at the LC. Background retrieval is not active by default, and must be explicitly selected by the user, since it changes the semantics of using the Web.

C. Privacy

The Selective Encryptor Customizer encrypts sensitive information that passes over insecure HTTP connections. Most e-commerce sites use secure connections for all transactions involving credit cards. However, many websites freely transfer other potentially sensitive information such as e-mail addresses, mailing addresses, and phone numbers over insecure connections. The Selective Encryptor uses encryption to protect that selective information from any host along the path from the RC to the LC. Since wireless networks are generally more prone to eavesdropping than wired networks, the encryption can be done at a location on the wired network before the data passes through the wireless network. The Selective Encryptor is even more beneficial when the RC-Server and the Web server are both in a trusted security domain. Note that the information will not be protected by the Customizer between the Web server and the RC, or between the LC and the client if they are on different hosts.

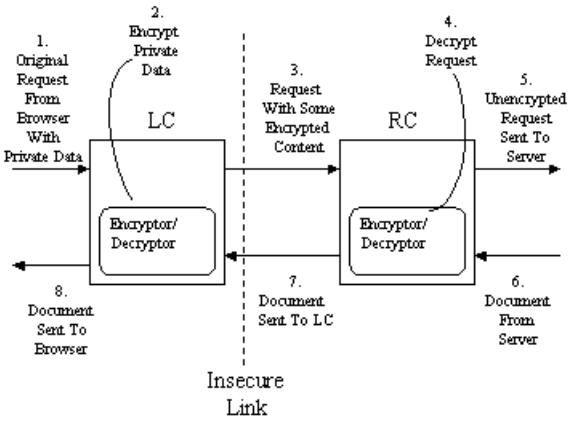

Fig. 10. The Selective Encryptor

As this is a content-based customization, the Selective Encryptor only encrypts requests and responses that it detects contain sensitive information. The user supplies strings to search for in text data including form submission, such as their mailing address. The LC can encrypt requests, and decrypt responses which were encrypted by the RC, while the RC decrypts requests that were encrypted by the LC and encrypts responses. Figure 10 shows how the Selective Encryptor is used to encrypt private data in Web requests, such as data from a form submitted by the user.

V. PERFORMANCE OF OUR WSC IMPLEMENTATION

The performance advantages derived from the ability to do remote customization can be negated if the underlying execution and communication mechanisms are slow. The use of Customizers introduces overhead because there are now two additional service points between Web client and Web server that operate in both directions. While we would like this

overhead to be low in absolute terms, the primary goal is that it should be low relative to typical Web transaction times.

We first conducted some simple Web experiments to determine typical Web transaction times from our site to some major popular sites. We are located on a university campus that has excellent Internet connectivity, as one of the major NAPs is on our campus, and our path to the NAP is high-speed. All our experiments were conducted at times when there was very low network traffic. We used high-speed PCs, based on 933 MHz Pentium III processors running Solaris x86 release 2.8, for clients, so that client delay would be low. Consequently, we expected the end-to-end Web transaction times to be relatively low and therefore good targets for comparison with Customizer overhead times.

We conducted three experiments (without using Customizers), where in each experiment, a client made 1000 requests directly to a Web server in an outside domain (pausing for 2 seconds between requests to assure quiescence). The three Web sites contacted were:

http://www.yahoo.com/,

http://www.suntimes.com/index/,

http://www.cnn.com/

These sites were selected because they are popular and they are located in three different geographic regions. (We used numeric IP addresses to avoid name-server delays; this is one of many examples of trying to reduce all sources of superfluous delays.)

The results of these experiments are as follows. The raw response times ranged from 126ms to as much as 24.5 seconds; however, the majority of response times were less than 500ms. To factor out anomalies, we discounted all response times longer than 1 second so that the average response times are somewhat more representative of reasonably good scenarios. (Recall that our goal is to simply determine good-case Web transaction times so that we can determine the impact of Customizer overheads.) Table 1 shows the average response times with 95% confidence intervals for the three Web sites.

TABLE 1

BASIC WEB TRANSACTION DELAYS

Web Site	Average Response Time (ms)
http://www.yahoo.com/	138 ± 0.8
http://www.suntimes.com/index/	404± 1.6
http://www.cnn.com/	475 ± 5.4

Yahoo had an average response time of 138ms, which was the best of the group. This is to be expected given that Yahoo is the geographically closest site to us. The two other sites are significantly more distant, and this is evident in the measurements, both of which averaged between 400-500ms. Consequently, if the total overhead introduced by Customizers is a small fraction of these average times, we can reasonably conclude that this overhead is acceptable. In a second set of experiments, we determined the basic overhead of a

Customizer by measuring the delay of a "null Customizer," i.e., a Customizer that does not modify the request or response, but simply forwards them. We used a test program that acts like a Web browser, and makes requests to a local Customizer-test environment, with Local and Remote Customizer Servers in place, a null RC installed on the RC-Server, and a null LC installed on the LC-Server.

In the test environment, the client, LC-Server, RC-Server and Web server each ran on a different machine, all of which were PCs based on 933 MHz Pentium III processors running Solaris x86 release 2.8 (the same used for clients in the previously described Web transaction experiments). All the machines were connected to an unloaded 100Mbps switched Ethernet LAN. The test browser program made 10000 total requests to the Web server's index page (i.e., a minimal 62-byte HTML page), pausing 50ms between requests to achieve quiescence between measurements.

Table 2 summarizes the results of these experiments. The average response time using Customizers was 6.5ms. Of this, we were able to attribute 4.8ms to actual overhead due to Customizers, as the average communication overhead between the client and LC-Server was 2.2ms, that between the LC-Server and RC-Server was 2.5ms, and the Customizer processing overhead was 0.1ms, leaving 1.7ms out of the 6.5ms for the non-Customizer portion of the Web transaction processing and communication. We also conducted a similar experiment, but without Customizers, and measured an average response time of 1.7ms, which provides experimental verification of our calculation.

TABLE 2

BASIC CUSTOMIZER DELAY RESULTS.

Measurement	Time (ms)
Response Time Using Customizers	6.5 ± 0.02
Client to LC-Server Communication Overhead	2.2 ± 0.01
LC-Server to RC-Server Communication Overhead	2.5 ± 0.01
Customizer Processing Overhead	0.1
Response Time For Direct Client To Server Requests	1.7 ± 0.02

While one might say that 4.8ms of overhead relative to 1.7ms for a basic Web transaction is high, this is only for the case where everything resides on a high-speed LAN with high-performance clients and Web servers, and the content being retrieved is minimal (62 bytes). What is important is that 4.8ms is small relative to human perception times, and is small relative to real Web transaction times where the delays are in the range of 100-500ms. This does not even take into account that this overhead is likely to be outweighed by the performance gains of using Customizers that actually do useful work (unlike null Customizers), such as reducing the amount of data being sent over the network or reducing the number of requests made to the end servers.

VI. Related Work

The most widespread method for adapting the Web to users' needs is to use a proxy. Traditionally proxies have been used primarily for security (firewalls and anonymity), and improving performance via caching [15]. However, there are a number of systems designed to use a single remote proxy for customizing the Web, with communication initiated through the browser's proxy mechanism. This includes image and video filtering, HTTP request modifications, HTML filtering, user interface improvements especially for small screens, remote caching, and support for disconnected operation and user-selected background retrieval [4, 8, 5, 6, 13]. Other systems have made use of the two-proxy (local and remote) concept, for such customizations as filtering, prefetching and intelligent cache management at the local proxy [13, 14].

Research that is closest to ours combines the use of proxies with mobile code to support dynamic downloading of filters to a remote proxy. Zenel uses both high-level and low-level proxies [25], and in [10] object migration is used to move an application running on a proxy to a new host in order to follow the movements of a mobile client. There are also customization systems that do not use proxies per se, but rather use more general mobile code mechanisms to support remote processing at arbitrary hosts, typically at the servers themselves [18, 21]. Going a step further, there are mobile agent systems that provide a highly generalized framework for code mobility [11, 9, 17, 20] that could be applied to Web customization.

An alternative to application-layer mobile code is to have code mobility in the routers, as in the Active Networks approach taken in [23]. Active Networks technology is complimentary to application-layer solutions such as proxy-based customization and Customizers, and is better suited to customization of network protocols rather than user-level data objects and application-layer protocols.

A related issue is adaptability, where information is provided to the client application, typically from the operating system, to help it adapt to changes in resource availability and network connectivity [1, 3, 16]. Some of these systems include applications using an adaptable interface, including adaptable protocols. Kunz and Black have introduced a proxy-based customization system that combines many of the above approaches [12]. They use both high and low-level proxies, system support for client software to be made aware of resource availability for adaptation, and the Objectspace Voyager mobile code system for dynamic distribution of code.

Our work differs from that of others in a number of ways. First, We have focused on a customization system designed specifically for the Web, allowing us to make a number of simplifying assumptions regarding the programming model, the user model, and the system design and implementation. Second, we use a very restricted and therefore more simplified form of mobile code, rather than providing a generalized mobile code solution which, while more powerful, is less practical and is more complex in terms of usability and security. Other unique features of our system include the use of an LC-Server that supports dynamic selection of multiple, simultaneously active, RCs. RCs can make use of LCs running on the LC-Server. We use a simple, callback-based

programming model for Customizers, and allow user-controlled selection of the Customizers, including the location of the RC, through a Web interface.

Our work is premised on the idea that Web applications would greatly benefit from the remote customization capabilities of our system. In fact, there exists a large body of research results verifying the benefits of remote Customization of Web data using proxies, mobile code, or some combination thereof. In [13] performance improvements of 25%-50% were reported for Web browsing over a cellular link. They used local and remote persistent caching, persistent connections between client and proxy as well as DNS prefetching to reduce round-trip delay, and prefetching of inline images to improve link utilization. Zenel showed a 50% reduction in delay using HTTP protocol and text content compression for files larger than 16K over a dial-up connection [25]. He also found a significant improvement in TCP throughput over error-prone connections using a version of Snoop TCP [2]. Loon and Bharghavan used user profile-based prefetching cooperating with a cache, in a system with both a local and remote proxy, and found that Web surfing waiting times can be reduced by a factor of 3-7 depending upon the time of day. According to [22], using remote processing to reduce the number of connections across a wireless link when browsing pages with images, can reduce response time significantly as the number of images in a page increases. For a page with 16 images, the average waiting time is reduced by approximately 30%. They also did experiments with remote compression and showed a 48% compression rate of .au audio files and a 94% compression rate for .mid audio files. In the PowerBrowser project, which uses a proxy filter to modify HTML pages into a special format to improve information retrieval time on a PDA with a stylus, the authors showed a 45% savings in time to complete tasks involving finding information on the Web [6]. Fox et al show a major reduction in end-to-end latency over a dial-up connection for image distillation that reduces the size and color-depth of images [8].

VII. Conclusions

We have presented a new Web customization architecture which is designed to be flexible, deployable, and user-friendly, and is tightly integrated with the existing Web model. The architecture provides a general customization framework that supports both content-based and system-based customization, and supports a variety of client-directed customization techniques.

The primary advantage of the WSC architecture is that it allows requested server content to be modified by having it processed by dynamically-deployed Customizers, selectively located between client and server. Because of their distributed operation by local and remote components, Customizers allow communication stream content and its transmission control to be effectively enhanced over selective portions of the communication path that require special considerations in terms of performance, reliability, and security.

We also presented three useful applications. We are currently gaining experience with using the applications. We

have demonstrated that the system overhead is low relative to typical Web transaction times, and thus the benefits of using Customizers are worthwhile.

REFERENCES

[1] David Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan. *System support for bandwidth management and content adaptation in Internet applications*. In Proceedings of 4th Symposium on Operating Systems Design and Implementation, pages 213-226, San Diego, CA, October 2000. USENIX Association.

[2] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy Katz. Improving TCP/*IP performance over wireless networks*. In Proceedings of the 1st MOBICOM, Berkeley, CA, November 1995.

[3] Vaduvur Bharghavan and Vijay Gupta. *A Framework for Application Adaptation in Mobile Computing Environments*. Proceedings of IEEE Compsac'97, November 1997.

[4] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. *An active transcoding proxy to support mobile web access*. In Proceedings of IEEE Symposium on Reliable Distributed Systems, 1998.

[5] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. *Application-specific proxy servers as HTTP stream transducers*. In 4th Intl. World Wide Web Conference, pages 539--548, December 1995.

[6] Buyukkokten, O., Garcia-Molina, H., Paepcke, A., Winograd, T. *Power Browser: Efficient Web Browsing for PDAs*. In Proceedings of CHI 2000.

[7] W. M. Farmer, J.D. Guttman and V. Swarup. *Security for mobile agents: Issues and requirements.* In National Information Systems Security Conference, National Institute of Standards and Technology, October 1996.

[8] A. Fox, S. Gribble, Y. Chawathe and E. A. Brewer. *Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives*. IEEE Personal Communications, Special Issue on Adaptation, August 1998.

[9] Robert S. Gray. *Agent Tcl: A transportable agent system*. In Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, December 1995.

[10] A. Hokimoto and T. Nakajima, "*An Approach for Constructing Mobile Applications Using Service Proxies*," Proceedings of the 16th International Conference on Distributed Computing Systems, May 1996.

[11] D. Johansen, R. van Renesse, and F. B.Schnieder. *Operating system support for mobile agents*. In Proceedings of 5th IEEE Workshop on Hot Topics in Operating Systems, Nov. 1994.

[12] Thomas Kunz and James P. Black, *An architecture for adaptive mobile applications*, Proceedings of Wireless 99, the 11th International Conference on Wireless Communications, Calgary, Alberta, Canada, July 1999, pp. 27-38.

[13] M. Liljeberg, T. Alanko, M. Kojo, H. Laamanen, and K. Raatikainen. *Optimizing World-Wide Web for Weakly-Connected Mobile Workstations: An Indirect Approach*. In Proc. 2nd International Workshop on Services in Distributed and Networked Environments (SDNE), pages 132--139, Whistler, Canada, June 1995.

[14] Tong Sau Loon and Vaduvur Bharghavan. *Alleviating the latency and bandwidth problems in www browsing*. In Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, December 1997. URL: http://timely.crhc.uiuc.edu/.

[15] A. Luotonen and K. Altis. *World-Wide Web proxies*. Computer Networks and ISDN Systems, 27(2), 1994.

[16] B. Noble, *System support for mobile, adaptive applications*, IEEE Personal Computing Systems, vol. 7, no. 1, p. 44-9, Feb. 2000.

[17] Peine H., Stolpmann T., *The Architecture of the Ara Platform for Mobile Agents*, In: Rothermel K., Popescu-Zeletin R. (Eds.), Mobile Agents, Proc. of MA'97, Springer Verlag, Berlin, April 7-8, LNCS 1219, pp 50-61.

[18] S. Perret and A. Duda. *Implementation of MAP: A system for mobile assistant programming*. In Proc. IEEE International Conference on Parallel and Distributed Systems, Tokyo, June 1996.

[19] *Java Servlet Technology Whitepaper*. http://java.sun.com/products/servlet/whitepaper.html. September 2000.

[20] M. Straßer, J. Baumann, and F. Hohl. *Mole - A Java Based Mobile Agent System*. In Proceedings of the ECOOP'96 workshop on Mobile Object Systems, 1996.

[21] A. Vahdat, M. Dahlin, T. Anderson, A. Aggarwal, "*Active Names: Flexible Location and Transport of Wide-Area Resources*," In Proceedings of the Second Usenix Symposium on Internet Technologies and Systems, Boulder, CO, October 1999.

[22] Y. Villate, D. Gil, A. Goni, and A. Illarramendi. *Mobile agents for providing mobile computers with data services*. In Proceedings of the Ninth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 98), 1998.

[23] David J. Wetherall, John Guttag, and David L. Tennenhouse. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*. In IEEE OPENARCH, April 1998.

[24] Bennet S. Yee. *A Sanctuary for Mobile Agents*. DARPA Workshop on Foundations for Secure Mobile Code, Monterey, CA, USA, March 1997.

[25] B. Zenel and D. Duchamp. *A general purpose proxy filtering mechanism applied to the mobile environment*. In

Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking, pages 248--259, Budapest, Hungary