A Web Middleware Architecture for Dynamic Customization of Content for Wireless Clients

Jesse Steinberg and Joseph Pasquale
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA 92093 USA
(1) 858-534-8604
{jsteinbe,pasquale}@cs.ucsd.edu

ABSTRACT

We present a new Web middleware architecture that allows users to customize their view of the Web for optimal interaction and system operation when using non-traditional resource-limited client machines such as wireless PDAs (personal digital assistants). Web Stream Customizers (WSC) are dynamically deployable software modules and can be strategically located between client and server to achieve improvements in performance, reliability, or security. An important design feature is that 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 implementation, and illustrate its use with three non-trivial, adaptive Customizer applications that we have built. We show that the overhead in our implementation is small and tolerable, and is outweighed by the benefits that Customizers provide.

Categories and Subject Descriptors

C.5.0 [**Computer System Implementation**]: General.

General Terms

Design, Experimentation, Performance.

Keywords

HTTP, Middleware, Proxy, Wireless, Mobile Code.

1. INTRODUCTION

Given the growing popularity of new wireless personal Web-access devices, users would like to be able to customize their view of the Web. This includes 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 may be compressed and possibly encrypted at some point before the wireless link and then decompressed and decrypted at the client. Provisions may also be made to deal with disruptions of important transactions, as well as masking short-term intermittent disconnections.

These types of customization should be able to dynamically adapt to changes in system conditions or 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 may outweigh this drawback.

In this paper, we present a new middleware 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 require such cooperation and distribution of functionality. For example, data compression (e.g., to reduce bandwidth requirements, and perhaps latency) requires that compression be done before the data crosses any relatively low-bandwidth links, and that decompression be done afterwards.

Another key point is that customizers provide client-specific customization of server content, effectively adapting the Web to new and different types 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 change are 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 remainder of this paper is organized as follows: In Section 2 we describe the WSC architecture. The details of how to use Customizers are described in Section 3. In Section 4 we illustrate some sample Customizer Applications. In Section 5 we present implementation issues, and in Section 6 we analyze the performance of our implementation. Section 7 details related work, and in Section 8 we present our conclusions.

2. THE WSC ARCHITECTURE

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. Customizers (and the entire server-support system) are written in Java, as will be explained further in Section 5.

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.

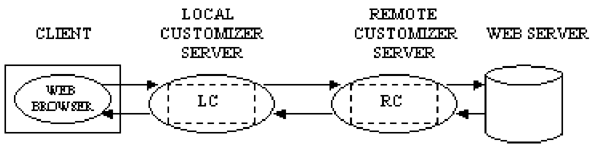


Figure 1. Local and Remote Components of a Customizer Running on LC and RC-Servers.

A simple example that illustrates the most basic capabilities of Customizers is shown in Figure 2, where an Image-Filtering Customizer is located on a base station to reduce image sizes, both to tailor it for the limited display of a wireless PDA and to reduce latency when transmitting the image over a bandwidth-limited wireless link. In this case, the local component does nothing, and the remote component does the actual filtering at the base station. This example illustrates the most basic capability of Customizers, that of strategically-located remote processing.

The reason for separating a Customizer into two components is that 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.

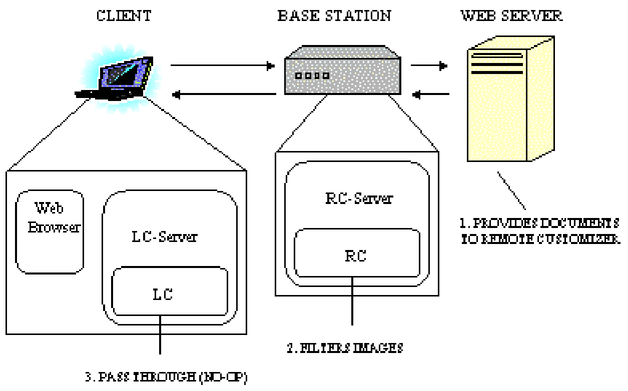


Figure 2. An Image-Filtering Customizer.

The RC generally performs location-dependent tasks that benefit from being near the server (or simply away from the client), such as compressing 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.

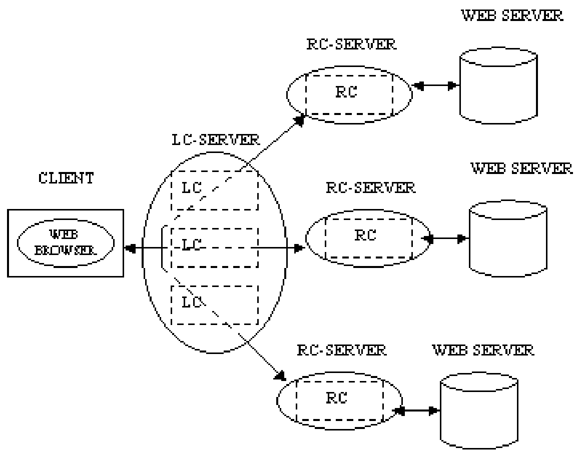


Figure 3. Using Multiple Customizers.

There will generally be many Customizers that are active simultaneously on behalf of a single client, each one being a separate (LC, RC) pair. 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 3.

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 only needs to have its proxy server settings configured to point to the LC-Server, and all of its requests will automatically be passed to the LC-Server, which can then pass those requests to Customizers. 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.

For a particular session of Web browsing, the same LC-Server is always used. 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. The LC-Server dynamically routes Web requests to different RCs by choosing an RC on a particular RC-Server based upon the URL of the request. This will be explained in Section 3.

3. USING 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.

3.1 Loading Customizers

Once the browser is configured to use an LC-Server as its proxy (described below), the user can *load* Customizers by clicking on special hyperlinks in Web pages. When the hyperlink is clicked, the Customizer is seamlessly loaded by the LC-Server. The LC-Server gets all the information it needs to load the Customizer from a file to which the hyperlink points, called a Customizer Meta File (CMF), which may reside on any Web Server. This information includes:

- The hostname of the machine running the RC-Server that will run the RC of the Customizer
- The name of a Java Archive (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 Domain of Applicability which specifies the sites to be acted upon by the Customizer
- Optionally, a URL for the Customizer's configuration page.

Figure 4a shows a web page with links to CMFs. Where and how the CMF is installed is described below. Figure 4b shows the process of loading a Customizer by clicking on a hyperlink to a CMF. When the link is clicked, the LC-Server intercepts the request and retrieves the CMF from the Web server. Once the LC-Server has received the CMF, it can download the LC, and send a message to the browser to inform the user that the Customizer was loaded. The next section describes how the LC-Server handles HTTP requests and passes them to Customizers.

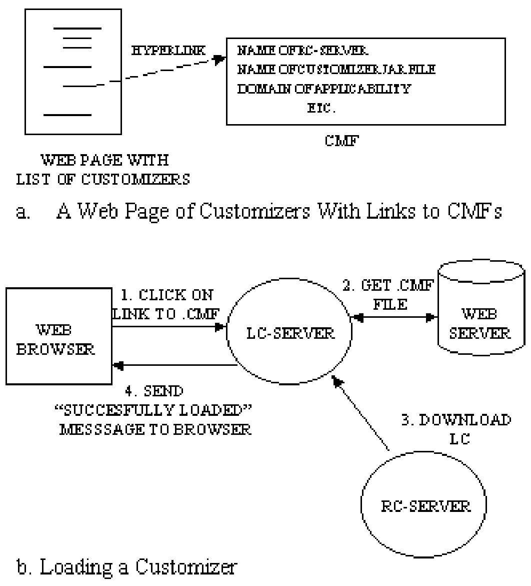


Figure 4. Web-based Customizer Loading.

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 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].

3.2 Handling Requests

Associated with each Customizer is a set of Web sites called its Domain of Applicability (DA). A Customizer will only operate on requests to (and responses from) sites in its DA. 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 5. 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, thus bypassing all Customizers (and defaulting to normal Web operation).

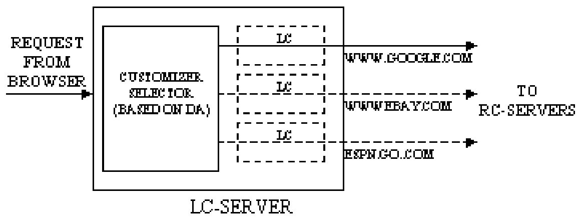


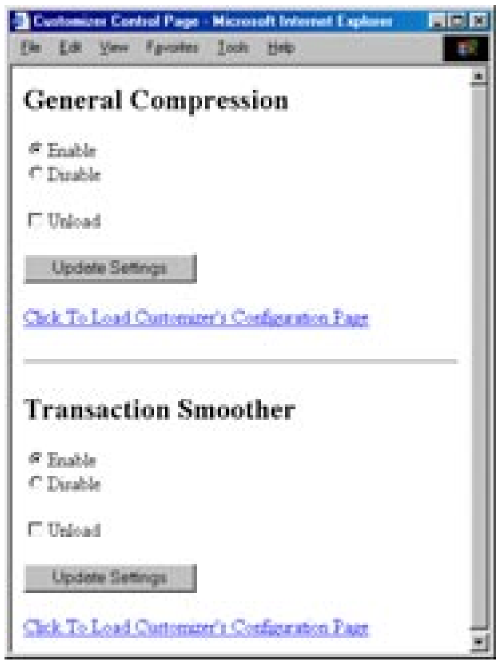
Figure 5. Selecting a Customizer based on DA.

In addition to helping select a Customizer, the DA also helps in protecting a client's privacy interests. For example, a client may want a particular Customizer to know about only certain requests. By matching the DA to the client's requirements, 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 favorite shopping sites to the LC-Server, with instructions not to allow any Customizers to handle requests to these sites. DA restrictions can also be used to prevent Customizers from being used to maliciously take over a portion of the URL space (also known as Web spoofing). Untrusted Customizers can be restricted so that they are only allowed to act on requests to URLs that are prefixed by the URL of the Customizer's CMF. Trusted Customizers can be given broader access to URLs.

3.3 Configuring Customizers

At any time, the client can select a special *control page* for a Customizer made available by the LC-Server. Figure 6 shows a control page after two Customizers, named General Compression and Transaction Smoother, have been loaded. The corresponding applications will be described in Section 4. This page has two major purposes. It can be used to directly control the use of Customizers, i.e., to *enable* or *disable* Customizers, or to *unload* them. Enabling/disabling toggles them on/off and unloading actually removes them from the LC-Server and RC-Server. 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 provided by the Customizer (and, probably, will 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. The Customizer control page is loaded by entering the host and port of the LC-Server into the browser's location area. For example, if the LC-Server is running on port 3000 on host customizer.test.edu, then the following URL would be entered into the browser's location area: "http://customizer.test.edu:3000/". (To simplify this for the user, a bookmark can be automatically set up to point to this page through an installation script when the LC software is installed.)


Customizer Control Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help

General Compression

[✓ Enable] Disable

☐ Unload

Update Settings

Click To Load Customizer's Configuration Page

Transaction Smoother

[✓ Enable] Disable

☐ Unload

Update Settings

Click To Load Customizer's Configuration Page

Figure 6. Customizer Control Page.

3.4 Installing a Customizer on an RC-Server

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. For each Customizer, a jar file containing the classes for the LC and RC must be placed in the *Customizer directory* of the RC-Server host. This directory defaults to a subdirectory of the directory from which the RC-Server was launched called *customizers*. An alternative directory can be provided as a parameter when the RC-Server is launched. The Web Server can then publish a CMF for each Customizer. As mentioned above, each CMF will contain the name of the RC-Server host, and name of the corresponding jar file that was placed on the RC-Server host.

3.5 Activating the Servers

The Java classes for the LC-Server and RC-Server are stored in jar files called "LCServer.jar" and "RCServer.jar" respectively. The RC-Server uses port 3001 by default, but an alternative can be specified as a command-line parameter. The RC-Server is loaded with the following command:

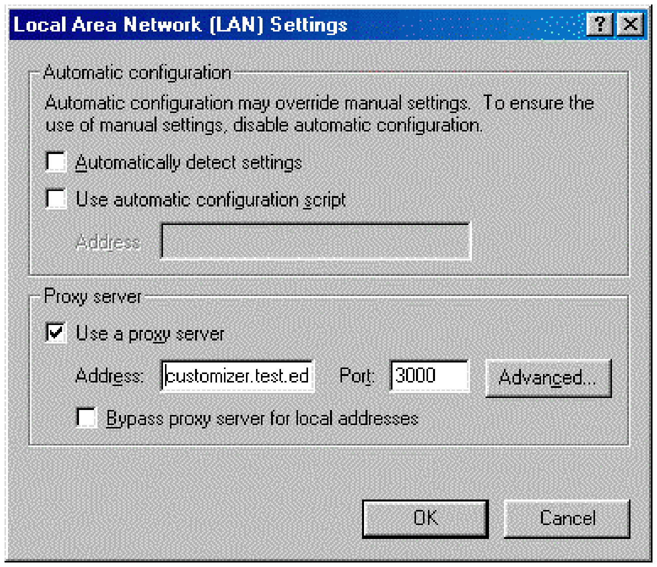
"java –jar RCServer.jar [-port <port#>]", where the port number parameter is optional.

The LC-Server is loaded with a similar command:

"java –jar LCServer.jar [-port <port#>]", the default port number is 3000.

Scripts can be used to simplify the loading of the servers without using the command line.

Once an LC-Server is started, in order to interface it to the browser, the user must enter its IP address and port into the browser's HTTP proxy configuration settings. Figure 7 shows Internet Explorer's proxy settings page (which is in the LAN Settings dialogue inside the "Connections" preferences tab), with the HTTP proxy being set to a fictional LC-Server running on a host called "customizer.test.edu" on port 3000. Netscape's proxy settings can be configured from the advanced preferences. Since the LC-Server acts as a proxy server, it will intercept and forward all of the browser's requests, so the user can continue to surf the Web normally after the browser has been configured, even if they are not using any Customizers.


Local Area Network (LAN) Settings

Automatic configuration

Automatic configuration may override manual settings. To ensure the use of manual settings, disable automatic configuration.

☐ Automatically detect settings

☐ Use automatic configuration script

Address:

Proxy server

☑ Use a proxy server

Address: customizer.test.ed Port: 3000

☐ Bypass proxy server for local addresses

Figure 7. Proxy Configuration Page.

4. CUSTOMIZER APPLICATIONS

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

4.1 Adaptive Compression

Two examples of adaptive compression Customizers are a General Compressor and an Image Filter. For the General Compression Customizer, the RC performs lossless compression on types of content that compress well, and the LC decompresses data compressed by the RC. This is beneficial when the RC is running a high-bandwidth, reliable connection to the Web Server, the LC-Server is on the client, and the client is connected via 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 which may be used to indicate to 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. Figure 8 shows the functioning of the General Compressor.

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.

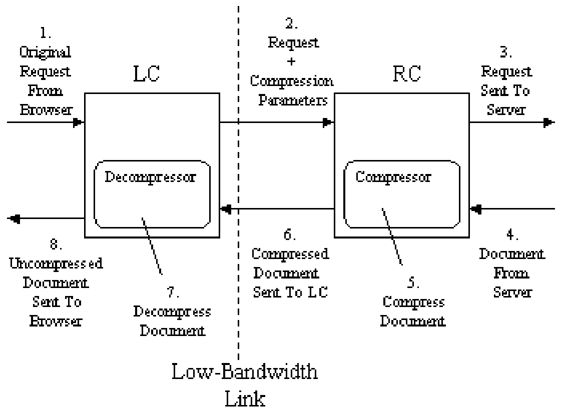

1. Original Request From Browser
LC
2. Request + Compression Parameters
RC
3. Request Sent To Server
Decompressor
Compressor
8. Uncompressed Document Sent To Browser
7. Decompress Document
6. Compressed Document Sent To LC
5. Compress Document
4. Document From Server
Low-Bandwidth Link

Figure 8. The Compression Customizer.

The RC handles the actual filtering. It provides four 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). This also requires updating image dimensions in HTML pages so that the images display properly
- Reducing the image color-depth by turning a color image into a grayscale image
- Converting images into formats that yield higher compression ratios
- Reducing the size of compressed images by compressing them at higher compression levels.

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, whether or not uncompressed or poorly compressed images should be converted into another format, and the compression level to use for compressed images. 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 bandwidth. When the response times speed up, or the frequency of downloaded images reduces, the LC can recommend that filtering be turned off entirely.

4.2 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 HTML pages. The user will then have to reload the page after connectivity is re-established.

Customizers can be used to mask such failures. Consider a scenario where the browser and LC are co-located on the same client machine, 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., an unreliable 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. If 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 re-established in a short time. If connectivity is re-established 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 re-established, 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.

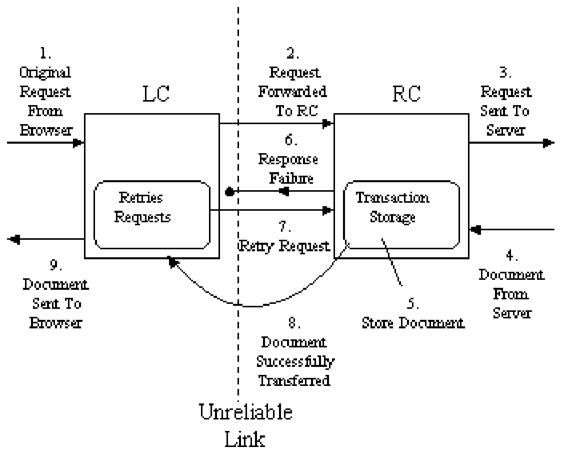

1. Original Request From Browser
LC
2. Request Forwarded To RC
RC
3. Request Sent To Server
6. Response Failure
Retries Requests
Transaction Storage
7. Retry Request
4. Document From Server
9. Document Sent To Browser
5. Store Document
8. Document Successfully Transferred
Unreliable Link

Figure 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 re-established, the user can bring up the Customizer's configuration page, which contains a list of all recorded transactions. This is useful for transactions that should not be repeated, such as transfer of money. As soon as connectivity is re-established, 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.

4.3 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 susceptible 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.

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.

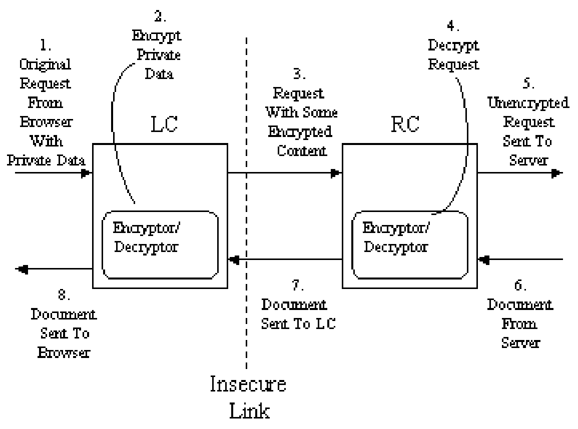

1. Original Request From Browser With Private Data
2. Encrypt Private Data
LC
3. Request With Some Encrypted Content
4. Decrypt Request
RC
5. Unencrypted Request Sent To Server
Encryptor/ Decryptor
Encryptor/ Decryptor
6. Document From Server
7. Document Sent To LC
8. Document Sent To Browser
Insecure Link

Figure 10. The Selective Encryptor.

5. IMPLEMENTATION ISSUES

We chose to implement the WSC architecture in Java because of the widespread availability (actual and 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 (which is an important feature of our implementation that 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 is another goal of the implementation. 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 an 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. When the LC or RC receives a request data buffer, it has the option of generating its own response buffer, or calling a method provided by the C-Server that will forward the request along and eventually return a response that was generated in the forward path. The response is returned back to the origin of the request by returning from the callback method with the response buffer as the return value.

6. PERFORMANCE

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.

6.1 Customizer Overhead Experiments

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 LC- and RC-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, providing 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 ± 0.10
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. Furthermore, this does not 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.

6.2 Adaptive Image Filter Experiment

The adaptive image filter experiment highlights the end-to-end performance benefits of having a Customizer provide adaptive customization to a wireless client. This experiment simulates the scenario depicted in Figure 2, in which a mobile client communicates with a base station over a wireless link, and the base station has a wired path to the rest of the Internet. We will assume that the wired path has consistently high bandwidth, whereas the bandwidth of the wireless link is lower and is variable over time. Since the quality of the adaptation depends upon how quickly the Customizer reacts to changes, it is beneficial to have it located just beyond the wireless link (e.g., near the base station).

The goal of the adaptive image filter is to maintain a constant transfer time despite variations in available wireless bandwidth. Consequently, when the client requests an image from a Web server during a period of low bandwidth, the image should be more compressed by the RC; while the image quality is reduced, the transaction time is also reduced to an acceptable level. However, when the bandwidth becomes higher, the RC should adjust by compressing less and thus sending a larger, higher-quality version. In either case, the image remains in a compressed image format that the client can read, thus making this filtering completely transparent (other than in performance) to the client and server.

To determine available bandwidth, when the LC is about to forward a request for an image, it also sends (using an HTTP header) the measured transfer time of the most recently received image to the RC. The RC uses this information to compute the last-known available bandwidth, which is then used to determine the level at which to compress the requested image (on its way back from the server). Of course, this means that if the bandwidth changes suddenly, the first transfer after this change will suffer because it is using old information. However, this will be corrected for in the next transfer, as the adapter will then be using the measured response giving the new bandwidth level. We implemented this simple adaptive algorithm mainly to demonstrate how an adaptive Customizer might work, and to illustrate its performance benefits. With more effort, this algorithm could certainly be improved.

In this experiment, we used three PCs connected via an isolated 100Mbps Ethernet switch. A 500MHz Pentium II PC runs a Web client program that periodically requests a 205K JPEG test image from a Web server. The LC-Server also runs on this PC. We simulated a variable-throughput wireless connection by running a throughput-regulating proxy on the same PC. The LC-Server communicates with the regulator, which forwards all requests to the RC-Server running on a 933MHz Pentium III PC. All responses returned by the RC-Server to the LC-Server are intercepted by the regulator, which controls the rate at which the data is sent to the LC-Server. The throughput from the regulator to the LC-Server changes periodically, as shown in Figure 11, simulating changes in the bandwidth of a wireless link. A Third 450MHz Pentium II PC ran the Web Server.

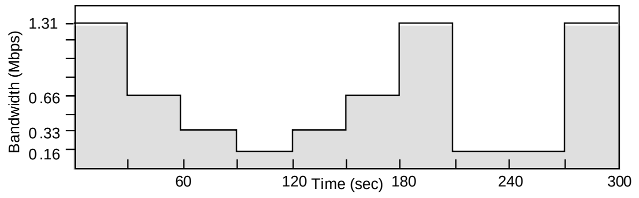


Figure 11. Variation in Available Bandwidth.

Figure 12 depicts the transfer times for the image requests made by the client during the course of the experiment. The figure is divided into rows, each row representing 1 minute of time. The rectangles represent image requests. The height of each rectangle indicates the bandwidth level during the time the request was active. This corresponds to the bandwidth as depicted in Figure 11. (Note that the y-axis is logarithmic in Figure 12, and that the time scale is stretched out when compared to Figure 11). The width of each rectangle represents the end-to-end duration of the corresponding request, from the time just before the client sent the request to the LC-Server, to the time that the client receives the image. For example, the first rectangle in the upper left corner of Figure 12 represents the first request made by the client. It starts at time 0, when the bandwidth is 1.31 Mbps, and takes approximately 2 seconds. The shaded rectangles represent requests that occur immediately after the bandwidth has changed. These will generally be wider if a reduction in bandwidth just occurred; the requests take longer because the adapter has not yet adapted to the change in bandwidth. So, for example, at 60 seconds, when the bandwidth reduces from 0.66 Mbps to 0.33 Mbps, the duration increases from about 2.7 seconds per request, to about 4.5 seconds for the first request after the reduction, and then reduces for the following requests as the adapter starts compressing more.

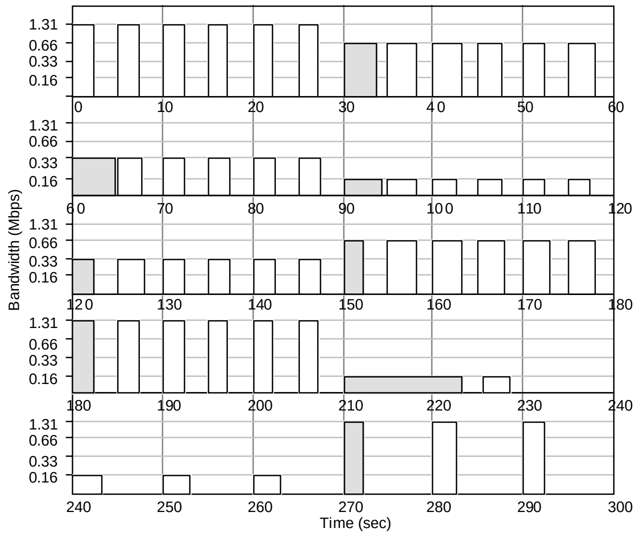


Figure 12. Experimental Results.

Except for the shaded requests, which are longer due to the lag in adaptation, the adapter yields steady transfer times across all requests. At the maximum bandwidth level, no image compression takes place at the adapter. For all other (lower) bandwidth levels, the adapter compresses the image to a level where the transfer time is roughly the same as it was at the previous bandwidth level. One complicating factor is that the adapter itself takes roughly 0.9 seconds to compress the test image. Thus, the transfer times for when the adapter is processing the image are a bit greater than those of the first 6 requests when no image processing occurs and bandwidth is at its maximum level. But with a faster processor, this overhead will be reduced, and so the benefits of adaptive compression scale with the speed of the RC-Server.

The shaded rectangles are thinner than their successors when the available bandwidth increases (at 120, 150, 180, and 270 seconds). This is again due to lag; the adapter is still compressing the image for a lower bandwidth, and hence the smaller, lower quality image is being transferred more quickly over the increased bandwidth. Once the adapter recognizes the increase in bandwidth, it reduces the compression ratio, so that the user can receive a higher quality image while keeping the transfer time steady. This effect is less noticeable at time 180 seconds, because the adapter stops compressing when it discovers that the bandwidth is at a maximum, hence the image processing overhead is no longer incurred.

The transfer time is about 13 seconds after the sharp drop-off in bandwidth from 1.31 Mbps to 0.16 Mbps at time 210 seconds, due to adapter lag. The adapter handles this sudden drop-off in bandwidth as expected. After the first request, it starts compressing at a high ratio, so that the transfer times for the next 4 requests at the same bandwidth are much faster. (To simplify the presentation of results, the time between requests was artificially increased to 15 seconds at this point in the experiment so that the 13 second transfer time would not overlap with responses of successive requests. However, the adapter works with overlapping requests, but only learns after the first one completes.)

7. 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 complementary 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-level 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 that, 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].

8. CONCLUSIONS

We have presented a new middleware system architecture for Web customization that 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 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 described how the system is used and presented a variety of useful applications. We are currently gaining experience with the applications, which is helping us better understand the range of optimizations enabled by the system. Finally, we have demonstrated that the system overhead is low relative to typical Web transaction times, and thus the benefits of using Customizers are worthwhile.

9. ACKNOWLEDGEMENTS

This work is supported by research grants from AFOSR, DARPA, and an NSF Research Infrastructure grant as part of the UCSD Active Web Project.

10. REFERENCES

[1] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan, "System support for bandwidth management and content adaptation in Internet applications," *Proceedings of 4th Symposium on Operating Systems Design and Implementation*, pp. 213-226, San Diego, CA, USA, October 2000. USENIX Association.

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz, "Improving TCP/IP performance over wireless networks," *Proceedings of the 1st MOBICOM*, Berkeley, CA, USA, November 1995.

[3] V. Bharghavan and V. Gupta, "A Framework for Application Adaptation in Mobile Computing Environments," *Proceedings of IEEE Compsac '97*, Washington, D.C., USA, August 1997.

[4] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul, "An active transcoding proxy to support mobile web access," *Proceedings of 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, USA, October 1998.

[5] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller, "Application-specific proxy servers as HTTP stream transducers," *Proceedings of 4th Intl. World Wide Web Conference*, pp. 539-548, Boston, MA, USA, December 1995.

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

[7] W. M. Farmer, J. D. Guttman, and V. Swarup, "Security for mobile agents: Issues and requirements," *Proceedings of 19th National Information Systems Security Conference*, National Institute of Standards and Technology, Baltimore, MD, USA, 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*, vol. 5, no. 4, August 1998.

[9] R. S. Gray, "Agent Tcl: A transportable agent system," *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, MD, USA, 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*, Hong Kong, May 1996.

[11] D. Johansen, R. van Renesse, and F. B. Schnieder, "Operating system support for mobile agents," *Proceedings of 5th IEEE Workshop on Hot Topics in Operating Systems*, Orca Island, WA, USA, May 1995.

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

[13] M. Liljeberg, T. Alanko, M. Kojo, H. Laamanen, and K. Raatikainen, "Optimizing World-Wide Web for Weakly-Connected Mobile Workstations: An Indirect Approach," *Proceedings of 2nd International Workshop on Services*

in Distributed and Networked Environments (SDNE), pp. 132-139, Whistler, Canada, June 1995.

[14] T. S. Loon and V. Bharghavan, "Alleviating the latency and bandwidth problems in www browsing," *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, Monterey, CA, USA December 1997. URL: http://timely.crhc.uiuc.edu/.

[15] A. Luotonen and K. Altis, "World-Wide Web proxies," *Computer Networks and ISDN Systems*, vol. 27, no.2, pp. 147-154, 1994.

[16] B. Noble, "System support for mobile, adaptive applications," *IEEE Personal Computing Systems*, vol. 7, no. 1, pp. 44-49, February 2000.

[17] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents," *Rothermel K., Popescu-Zeletin R. (Eds.), Mobile Agents, Proceedings of MA '97*, pp. 50-61, Springer Verlag, Berlin, Germany, April 7-8, 1997, LNCS 1219.

[18] S. Perret and A. Duda, "Implementation of MAP: A system for mobile assistant programming," *Proceedings of IEEE International Conference on Parallel and Distributed Systems*, Tokyo, Japan, 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," *Proceedings of the ECOOP'96 workshop on Mobile Object Systems*, Linz, Austria, July 1996.

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

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

[23] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," *Proceedings of IEEE OPENARCH*, San Francisco, CA, USA, April 1998.

[24] B. 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," Proceedings *of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 248-259, Budapest, Hungary, September 1997.