Responsive Interaction for a Large Web Application

The Meteor Shower Architecture in the WebWriter II Editor

Arturo Crespo
Stanford University
crespo@cs.stanford.edu

Bay-Wei Chang
Xerox Palo Alto Research Center
bchang@parc.xerox.com

Eric A. Bier
Xerox Palo Alto Research Center
bier@parc.xerox.com

Abstract

Traditional server-based web applications allow access to server-hosted resources, but often exhibit poor responsiveness due to server load and network delays. Client-side web applications, on the other hand, provide excellent interactivity at the expense of limited access to server resources. The WebWriter II Editor, a direct manipulation HTML editor that runs in a web browser, uses both server-side and client-side processing in order to achieve the advantages of both. In particular, this editor downloads the document data structure to the browser and performs all operations locally. The user interface is based on HTML frames and includes individual frames for previewing the document and displaying general and specific control panels. All editing is done by JavaScript code residing in roughly twenty HTML pages that are downloaded into these frames as needed. Such a client-server architecture, based on frames, client-side data structures, and multiple JavaScript-enhanced HTML pages appears promising for a wide variety of applications. This paper describes this architecture, the Meteor Shower Application Architecture, and its use in the WebWriter II Editor.

1 Introduction

1.1 The WebWriter application builder

The *WebWriter* system [Crespo96] supports the construction of simple interactive web applications without the need to learn HTML or CGI programming. Modeled after HyperCard [Apple87], WebWriter allows the user to build an application as a stack of pages, where each page can contain text, images, buttons, and other form elements, as well as content computed on the fly by executing scripts. The user constructs the layout of each page of an application using the *WebWriter II Editor*, an interactive editor that runs in any browser that supports frames and the JavaScript language [Netscape96a].

The user adds application behavior using the WebWriter II Editor by writing scripts that will be run either on the server or in the browser. Users without programming experience can add behavior by selecting a built-in program and filling in details for that program. For example, the user can select the built-in file listing program and fill in a form to specify how to determine which files to list.

In addition to the Editor, the WebWriter system includes the *WebWriter Page Generator*, a server-based CGI service that creates new pages as a WebWriter-built application runs. Because they use the Page Generator, applications produced by WebWriter run as CGI programs on a web server and hence can be used from many platforms and in many web browsers.

1.2 Increasing interactive performance

The original WebWriter Editor was a CGI program so that every interaction with the user had to go to the server for processing. The interactive speed of the program was poor due to network delays, startup time of the server-side script, and whole screen redraws at the client after each interaction. In addition, this solution was not scalable: as the number of users of the editor increases, the server becomes a bottleneck. This paper describes the architecture of a new version of the editor (the WebWriter II Editor) that overcomes these limitations. In this architecture, which we call the *Meteor Shower Application Architecture*, both the web browser and the web server collaborate in the execution of the WebWriter II Editor. Operations that need high interactive speed are performed in the web browser using JavaScript, while the server executes only the operations that need server resources or that otherwise cannot be performed by JavaScript in the browser.

The rest of the paper is organized as follows. First, we review related work. Then, to give context to the architecture discussion, we present the user interface of the WebWriter II Editor. We then describe what happens behind the scenes during a typical session with the editor, from starting the editor, to loading and modifying a web page, to finally saving the page. Having described the way the WebWriter II Editor works, we generalize these ideas and introduce the Meteor Shower Application Architecture. Finally, we discuss the advantages and disadvantages of the model and give our conclusions and plans for future work.

2 Related Work

There are many systems that divide an interactive application between a web server and a web browser. One way to do this is to use a Java applet [JavaSoft96a, Arnold96]. In this case, very general programs written in the Java language are downloaded to a browser where they can interact at high speed with the user. We chose JavaScript over Java in the WebWriter II Editor for several reasons, including:

1. Browsers can already display formatted HTML. We did not want to duplicate this functionality in Java. In the first place, it would be more work. In the second place, by using the browser's formatting we take advantage of any improvements in that formatting without having to update our code. Finally, for users who want to preview their HTML page in a particular browser, our implementation allows them to do this just by running WebWriter in the browser in question.
2. We anticipated that building our control panel components as fragments of HTML would be less work than building them as calls on the `java.awt` toolkit (Java's Abstract Window Toolkit) [JavaSoft96b] or the subArctic user interface toolkit [Hudson96].
3. Java applets must specify a fixed rectangle as their size. We wanted to allow the user to resize the

WebWriter editing region just by resizing the browser. This is easily done using frames.
4. By using JavaScript, we avoid the need to compile our code as we change it, so we can try out new versions of WebWriter very quickly.

Our architecture is similar in some respects to that used by the Krakatoa Chronicle [Kamba95]. Like the WebWriter II Editor, the Krakatoa Chronicle downloads a document (in this case a set of newspaper articles) to the browser which is then formatted at the browser for reading. Unlike the Krakatoa Chronicle, our system uses the native formatting capabilities of the browser, is implemented as a set of JavaScript-containing HTML pages, that are loaded into frames on demand.

Also similar is Netscape's PowerStart [Netscape96b], a multiple page, multiple frame JavaScript application for creating a home page. The constructed page, based on a small set of templates, is saved as a set of preferences in a browser cookie, and is recreated from that cookie on subsequent visits. Unlike PowerStart, the WebWriter II Editor provides direct manipulation editing, can create general web pages that can include forms and behavior, and uses the server for file operations and large processing tasks.

Other ways to provide interactive applications accessible from the web include helper applications and plug-ins, using, for example Mosaic CCI, the Netscape plug-in API, or Microsoft Active X. As with Java applets, we rejected these methods because we wanted to take advantage of the HTML formatting capabilities of the browser itself. In addition, plug-ins and helper applications must, in general, be written for a particular platform or browser; we wanted a system that would work on many browsers and platforms.

3 The WebWriter II Editor user interface

Before describing the architecture, we briefly present the main user interface elements of the new WebWriter II Editor. Figure 1 shows a typical screen.

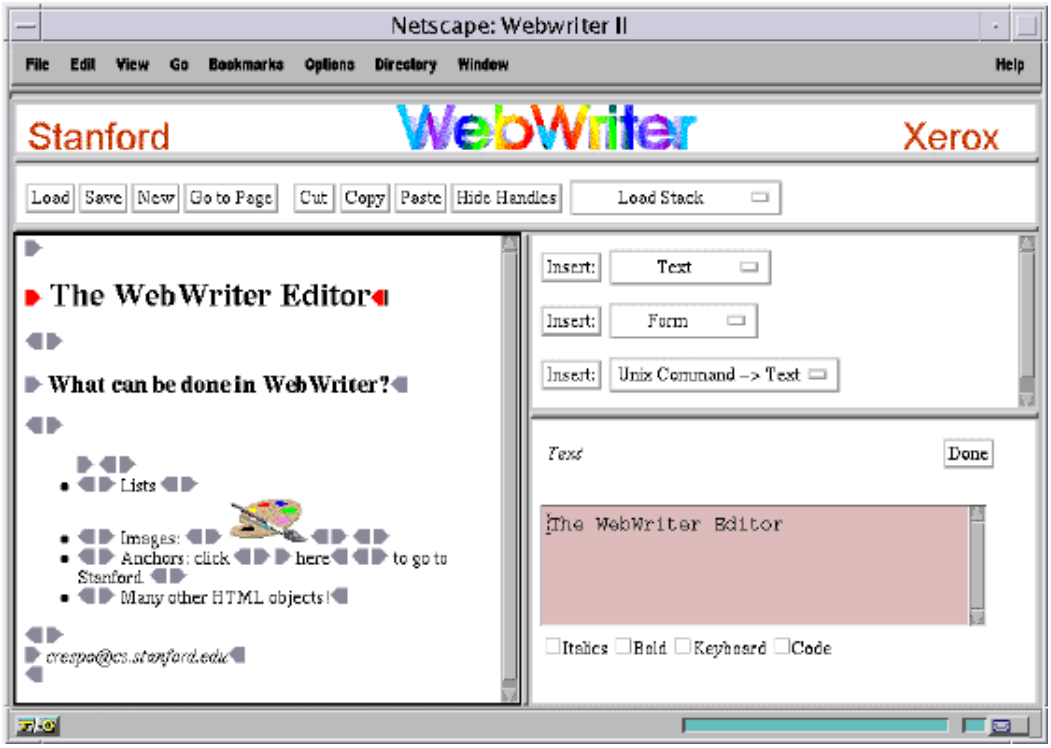

Figure 1: The WebWriter II Editor.

The editor consists of five frames tiling the browser window (see Figure 2). The *top level page* is invisible to the user; it contains the frameset (the HTML description of the sizes and positions of the five frames inside the browser window), the global JavaScript functions and data structures of the editor. The *title frame* holds the WebWriter logo. The *preview frame* contains the page that is being edited. The *general controls frame* provides file and stack operations and cut/copy/paste editing. The *object insertion controls frame* contains controls for inserting HTML elements. We refer to an HTML element in the preview frame as an "object." The *object properties frame* contains commands that are specific to the currently selected object.

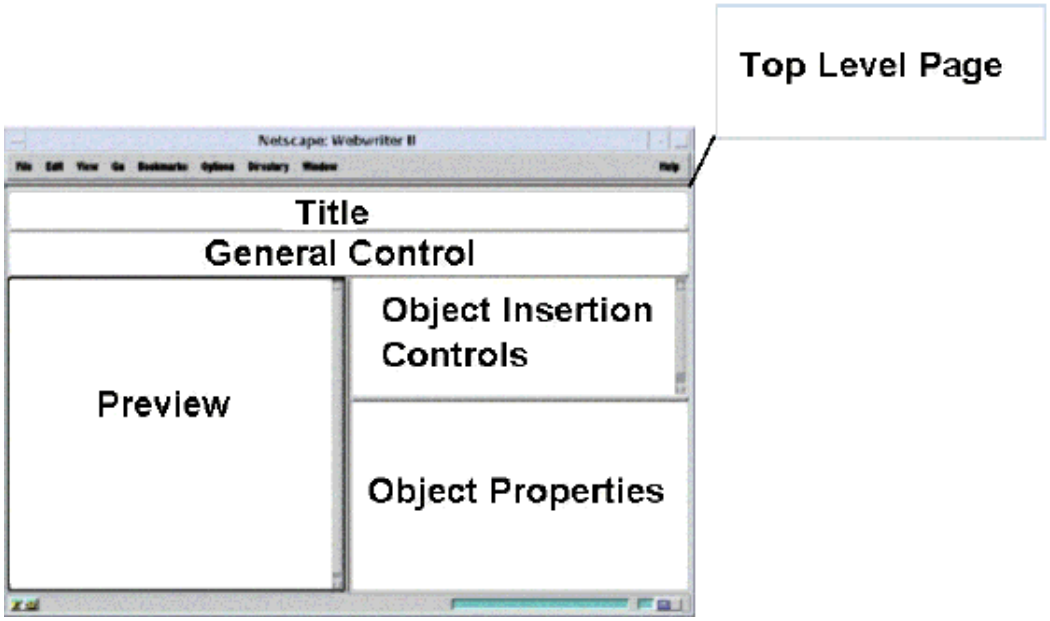

Figure 2: WebWriter II Editor frames.

In editing mode, the WebWriter II Editor displays the current page as interpreted HTML together with additional images, called *handles*, as shown in Figure 3. Handles are used to select an object; red handles indicate the currently selected object, and the blinking black bar next to the red handle is the insertion point. Selecting an object causes that object's properties to appear in the object properties frame, where they can be examined and changed. To insert an object, the user selects it in the insertion control frame and fills in its properties.

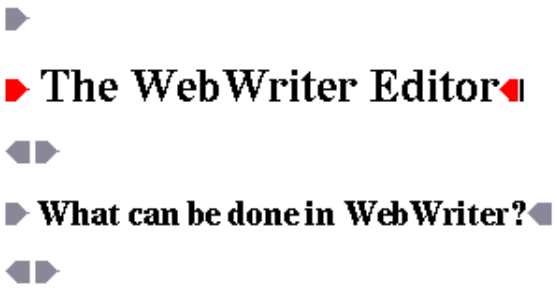

Figure 3: Handles (grey and red shapes) and the insertion point (vertical black bar to the right of the word "Editor").

There are many more facilities available in the WebWriter II Editor, including those for copying and pasting HTML, managing multiple page applications ("stacks"), and specifying behavior to execute when buttons are pressed on the page. For a detailed description of the original WebWriter Editor from the user's point of view, see [Crespo96].

4 The WebWriter II Editor architecture

The WebWriter Editor was re-designed in order to improve its interactive performance and to reduce screen clutter. As mentioned earlier, the original WebWriter Editor was implemented as a CGI script, in which every handle selection and button press was handled by the server. The user then had to wait for network travel, CGI startup, and complete re-layout and redrawing of the browser window. Since the WebWriter Editor was designed as an interactive, direct manipulation application, nearly every click of the mouse incurred this delay. Exacerbating the situation is the use by WebWriter Editor of many control elements surrounding the actual page elements being previewed -- handles and insertion points approximately tripled the number of non-text elements involved in layout and display. Even when WebWriter was running on a local web server on a very fast machine, the delay caused by a simple interface operation (selecting a handle, for example) was still several seconds long. Although several seconds is acceptable for operations that users expect to require some computation, this is much slower than the near-instantaneous response for interface-level operations in typical non-web graphical applications.

To make the WebWriter II Editor more usable, we focused on improving response time for interface operations, as well as improving the interface itself. We accomplished this in three parts: dividing processing, as appropriate, between a CGI script running on the web server and JavaScript functions running on the client browser; segmenting the interface into individually reloadable pages using multiple frames; and replacing images in place to reflect changes in state. The result is an editor in which response times for many operations are nearly instantaneous, and are comparable to those of standalone, non-web applications.

4.1 Selectively dividing processing between server and client

Browser scripting languages like JavaScript enable dynamic behavior without the overhead of traffic over the network. The new WebWriter II Editor was designed to use JavaScript to provide fast interactive behavior, resorting to the overhead of a CGI call only when server resources are needed, or when JavaScript cannot reasonably provide the behavior required. For example, computationally intensive operations may be technically feasible in JavaScript but run very slowly. In that case, the overhead of a CGI call (including network traffic and page redisplay) is worth the savings in processing time.

Of the 23 modules composing the WebWriter II Editor, four of the modules are CGI scripts written in the Python programming language [Watters96] and run in the server. The remaining 19 modules are HTML pages enhanced with JavaScript. Only five of the HTML modules are active at once, one in each of the WebWriter frames.

The CGI modules provide server-side services such as loading files, parsing HTML, saving files, and setting up the environment for the HTML modules at start up. The JavaScript modules handle displaying the edited page in the preview frame, selecting the current object, editing and insertion of HTML objects, and copying and pasting of objects.

In the following sections, we will show how processing is directed to the server and to the client as these basic tasks are performed: starting the WebWriter II Editor, loading and saving an HTML pages, and displaying and modifying the page.

4.1.1 Startup: Using the server to create the HTML environment

The user starts the WebWriter II Editor by invoking a CGI script at the server. The server-side CGI script creates an HTML page with three components: global JavaScript functions, calls to build the JavaScript global data structures, and the definition of the frameset, as shown in Figure 4. The global functions provide an interface to the global data structures, and provide common functionality needed by all modules. The global data includes the *document tree*, which holds the elements of the page that the user is editing, and global status information such as the position of the insertion point. The frameset defines the position and properties of the frames, as well as the URLs of their initial contents.

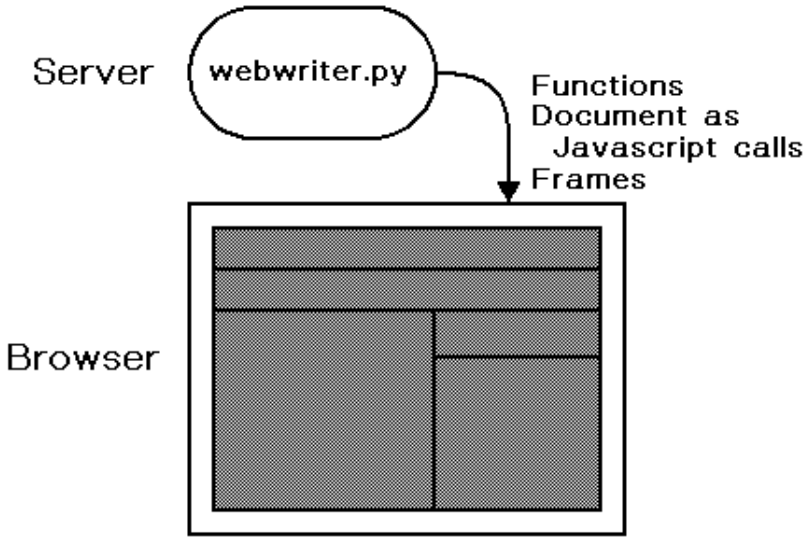

Figure 4: The server downloads functions, global data structures, and the component frames to the browser.

When the browser receives the HTML page generated by the server, it interprets the page by running the JavaScript function definitions, creating the JavaScript document tree and storing it at the top-level browser window. Then, it creates the frames and requests from the server the content of each frame, starting a "meteor shower" of HTML pages from the server to the browser, as shown in Figure 5.

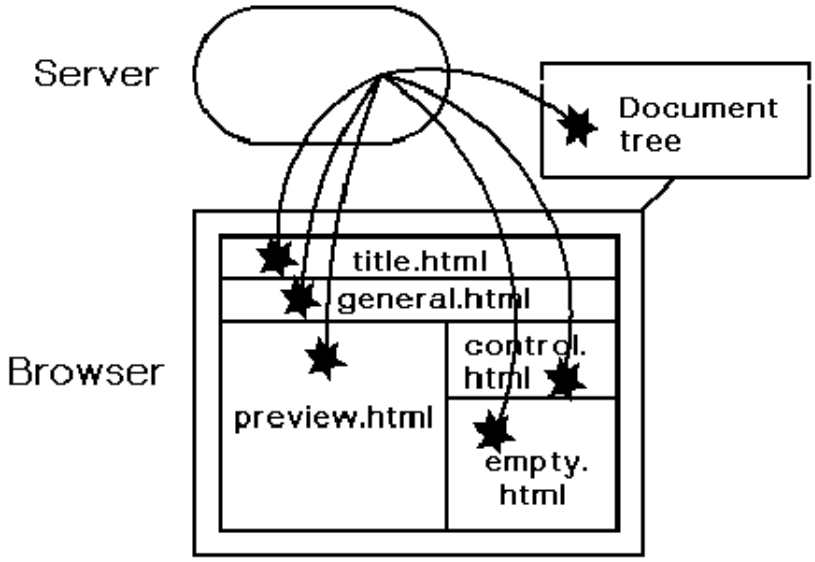

Figure 5: The server startup meteor shower.

The HTML page sent to a frame could be static HTML (such as the one used in the title frame) or an HTML page that includes JavaScript code. Pages with JavaScript code can collaborate with one another via global data structures and functions placed in the top level page of the browser. For example, the HTML page loaded in the preview frame contains a script that translates the document tree stored at the top window level into an HTML representation with handles.

4.1.2 Loading and saving pages: Using the server to access files

Figure 6: The general controls frame.

The general controls frame usually contains a module, `general.html`, for operations such as loading and saving files or cutting and pasting objects. This module defines routines to perform these commands and then describes the buttons in HTML. Each button contains a small piece of JavaScript that calls the associated function. For example, `general.html`, includes these two pieces, in JavaScript and HTML sections, respectively:

```
function Load() { ... }
...
<input type="button" value="Load" onClick="return Load();">
```

When the user clicks on a button, the JavaScript code associated with it is executed. The code can either execute locally, or it can ask the server to perform some service. For example, the "Hide Handles" button executes locally; first, a global variable is modified to change the display mode and then the preview frame is redisplayed to reflect the new mode.

The "Load" button is an example of an operation that requires the help of the server. We need to access the server for this operation because files are located in a server accessible file system. When the user clicks the "Load" button, the browser pops up a dialog box asking the user to supply the URL of a file to load. The information entered by the user is sent to a CGI script that runs on the web server.

The server tries to read and parse the URL specified by the user. If this succeeds, the server sends a new frameset as in the startup process; but this time, instead of sending an empty document tree, it sends a representation of the document tree for the requested document. In fact, because the server is a Python program and the client is running JavaScript, the server encodes the document as a set of nested JavaScript calls to be interpreted by the client. These calls look like this:

```
tree =
  new CreateChild(new CreateObject(
    new CreateState('h1','h1'),
    '<h1>', '</h1>',
    new CreateChild(new CreateObject(
      new CreateState('text', '', 'text', 'One Header', 'italics', 0, 'bold', 0, .
      'One Header', null, null,
      'text'
    )),
    "h1"
  ));
```

where each CreateObject call adds a new object to the tree, and each object, in turn, may have children. The fragment above builds the tree for the document "`<h1>One Header</h1>`". The reader need not understand this code in detail, but can simply note its nested form.

Saving a page is similar to loading a page. The user clicks in the Save button, which causes a JavaScript routine to execute. The script asks the user for the URL in which the file will be saved. Then, the document tree is transformed into standard HTML and, with the URL, is sent to the server. The server does two operations when saving a file. First, it translates the URL into a filename, and then it stores the HTML as a file at that location.

4.1.3 Displaying the HTML page: Using the client to construct modified pages

As described in the previous section, the document tree is a JavaScript data structure built when a page is loaded into the editor and stored in the root window. The preview frame generally contains module `preview.html`, which has three parts:

1. Definitions of JavaScript functions to walk the document tree and translate it into HTML that includes editing handles around each object. Using a tree data structure to represent the document improves program speed, because traversing this structure is faster than processing a linear string of characters in HTML format.
2. Definitions of JavaScript functions that are called when the user clicks on a handle.
3. A call to the JavaScript functions of part 1, which actually causes the new HTML to be written into the preview frame.

So for example, `preview.html` includes:

```
function DocumentToHTML(framedoc, userdoc, showHandles) { ... }
function OnClick (nodeID, objectPropURL) { ... }
...
DocumentToHTML(document, parent.userDocument, parent.showHandles)
```

and the generated HTML includes handles such as:

```
<a ... onClick="OnClick(264, 'text.html');">
<img src="left_handle.gif">
</a>
```

where the number "264" identifies the object being selected, and "text.html" is the name of the module to load into the object properties frame in order to edit the newly selected object.

After the user makes an edit to the document, the screen is redisplayed by calling the JavaScript `reload()` method on the preview frame. This updates the display without requiring any significant interaction with the server (because `preview.html` is cached at the browser and the document tree is converted to HTML as the browser interprets `preview.html`).

4.1.4 Selecting an object and modifying its properties: Using the client for interactive response

The first step for modifying an object is selecting it; to do this, we click on the handle that surrounds the object. This click triggers a JavaScript function that highlights the handles in red and updates global data structures to reflect the new current selection. The JavaScript code also loads into the object properties frame the appropriate HTML file for the class of the selected object.

The selection of an object is a small-scale meteor shower. Figure 7 shows how the process is initiated in the preview frame, requesting the server to send the appropriate HTML file to the object properties frame.

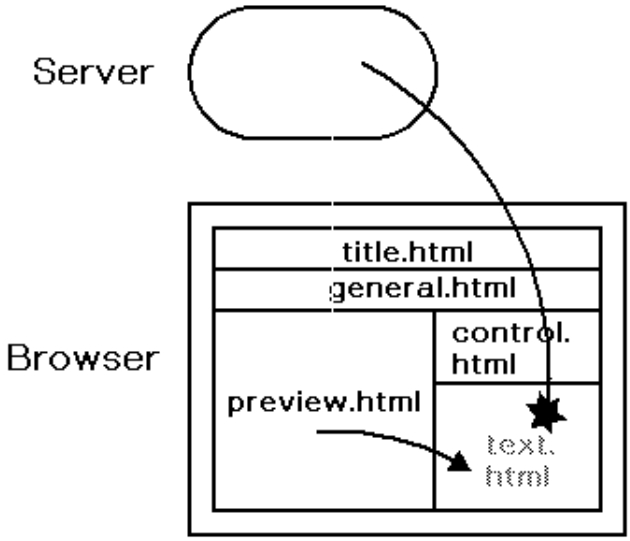

Figure 7: Editing an object.

The final step of modifying an object is changing its properties. This is done through an interface that is object-specific. Similar objects can share the same interface, but the programmer can develop specialized interface for some objects. For example, the text object has a specialized interface that allows the user to insert text and at the same time change the appearance of the text (see Figure 8).

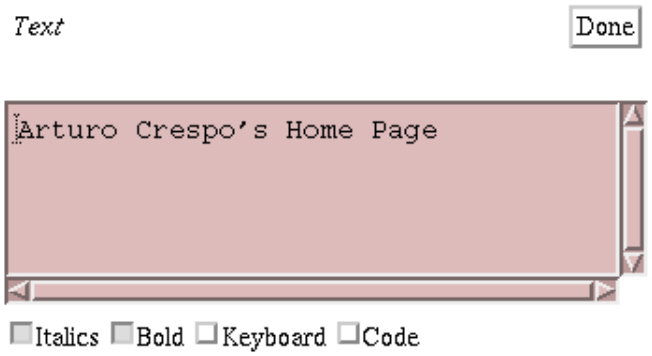

Figure 8: Text object interface.

The text object interface is a form that is initialized with the current properties of a text object. The user can modify those properties and then click the Done button. This triggers a client-side update of the document tree, generation of the new document view by reloading the preview frame, and replacement of the contents of the object properties frame with a blank page.

4.2 Segmenting the interface

The multiple frames that make up the WebWriter II Editor interface serve several purposes. Most importantly, frames separate the interface into areas that can be updated independently. One part of the interface can change by reloading its page without requiring other parts to be reloaded as well. For example, clicking on a handle to get its object properties causes a new page to be loaded in the object properties frame, but requires none of the other frames to reload. Reloading the minimum necessary is an important factor in reducing response times during user interaction.

Frames also play a role in making the editor more usable in realms other than responsiveness. Since reloading a frame usually causes it to become blank before the new page is displayed, the frames which do not change provide continuity of context. The editor behaves more like a standard desktop application which has selective control over what changes in the interface. In addition, placing the preview area in its own frame allows very long pages to be edited conveniently. Since the frames scroll independently, the preview frame can be scrolled without affecting the layout of the other controls in the interface.

Finally, from an implementation point of view, segmenting the interface modularizes the program. Each possible "state" of a frame is a JavaScript-enhanced HTML file. As illustrated in previous sections, active modules (those that are currently loaded in the frames) communicate with one another via global data, and cause other modules to be loaded and unloaded into the interface.

4.3 Replacing images in place

Making a new selection or changing the position of the insertion point are extremely common operations when editing a document, and thus should be performed as fast as possible.

Handles in the preview frame are grey if unselected, and red if selected. Clicking on a handle simultaneously selects it and deselects the old selection. In the original WebWriter Editor, this required a server round-trip to redraw the page appropriately. In the JavaScript-enhanced WebWriter II Editor, this could be handled by the client recreating the page. However, we made use of an even faster technology that exists in the current Netscape browser: replacing images of the same size in place, without a reload. This allows the highlighting to occur with no perceivable time lag.

The technique for displaying the insertion point also benefited from this technology. The original WebWriter Editor placed a radio button at every valid insertion point within the previewed page. Radio buttons are controlled by the browser, so selecting one is essentially an instantaneous operation that does not require a trip to the server. However, making all possible insertion points visible as radio buttons clutters the display considerably. Additionally, it was sometimes confusing to have radio buttons serving as insertion points mixed in with bona fide uses of radio buttons in the web page being constructed.

To minimize clutter, this scheme was replaced with one in which possible insertion points are no longer explicit. Instead, the selected element determines the insertion point. Now when the user clicks on a handle, that handle and its companion handle are replaced by red handles; in addition, one of those handles has a small bar next to it indicating the position of the cursor. To make the cursor more visible and also more like a traditional text editing cursor, we used animated GIFs to make it blink (one of the only tasteful uses we’ve seen of blinking on a web page).

5 The Meteor Shower Application Architecture

The architecture of the WebWriter II Editor -- multiple frames collaborating with one another via a browser scripting language and with the web server via CGI scripts -- can produce web applications that "in spirit" remain server-based, yet are highly responsive. This model we call the Meteor Shower Application Architecture, after the meteor shower of pages that the initial CGI script places into the browser’s frames.

The Meteor Shower Architectures relies in the functionality of the client browser and the server. The client browser provides processing for all interface operations, thereby ensuring short interactive response times. Since a Meteor Shower application is segmented into many frames, collaboration occurs among the frames as user operations in one frame cause other frames to update. Having many frames also limits updates to only those parts of the interface that require updating, potentially a significant savings because reload times can be relatively lengthy compared to the timeframe of individual interactions. Furthermore, the client can use in-place replacement of images to indicate state changes, wholly bypassing the need to reload or regenerate pages.

The server has three functions. First, it sets up the state of the client. Second, it provides the client with the programs it needs to run. And, third, it provides additional functionality (via CGI scripts) to the clients for operations that are either inefficient or impossible to do at the client side. This includes access to server resources as well as computations inappropriate for relatively slow scripting languages.

The distribution of work in the Meteor Shower Architecture has the following potential advantages and disadvantages.

5.1 Advantages

- *Improved performance.* Because the code dealing with user interaction is in the client, we don't incur any network delay communicating with the server. Additionally, because each client handles most operations locally, the contention for the server is not as severe as in the case of the pure CGI approach.
- *Economics*. There are two economic perspectives involved. First, by using the server for only the operations that are inefficient or impossible to perform on the client, we reduce the cost of maintaining the server. Second, by reducing the amount of communication between the browser and the server, we reduce the cost of communicating through the network.
- *Scalability*. The Meteor Shower Architecture makes supporting a large number of clients easier than a traditional CGI approach. Again this is related to the available resources of the server and the bandwidth of the connection between the server and the browser.
- *Faster development*. The Meteor Shower approach can allow for faster development than writing applets since it can make use of the browser's built-in capabilities, primarily the capability to display HTML. In the case of the WebWriter II Editor, not only does the browser handle layout of all control areas, it also handles layout of the preview of the HTML page being edited.
- *Debugging*. Because only a small number of modules are running at the same time, and there is a very clear interface between them, finding a faulty module is easy. Additionally, the use of an interpreted language allowed us to test the modifications faster than with a compiled language.
- *Security*. A well defined CGI interface the server and the browser, allows a secure environment to be maintained at the server side. Similarly the security safeguards in the browser maintain a secure client environment.
- *Caching*. Because browsers cache pages, both the appearance of user interface components (as HTML) and their behavior (as JavaScript) is cached, reducing still further the demands on server and network load. The client expect to incur a network delay for loading the HTML pages only the first time they are accessed; afterwards the pages should come from the cache.

5.2 Disadvantages

These advantages trade-off with problems arising from the distributed and interpreted nature of the Meteor Shower Architecture.

- *Intellectual property*. All of the JavaScript code is shipped to the client, potentially giving away a significant portion of the source code. A competitor may copy this code and do reverse engineering to write the missing server-side code.
- *Complexity*. The Meteor Shower Architecture is inherently more complex than the CGI model as it includes the challenge of a distributed architecture.
- *Dependence on caching*. As noted above, the HTML modules of this architecture are cached. This can obstruct software development if the client holds on to a stale version of a module the developer is trying to modify and test. We work around this problem by using the cache flushing capabilities of our browser.

6 Future work

We plan to incorporate the Meteor Shower Architecture into the WebWriter Page Generator. This will allow programmers to quickly develop powerful, highly interactive web applications. For example, such a system could allow a programmer to develop an application like WebWriter using WebWriter itself.

We have improved the interactive speed by reducing the number of times that the application needs to go to the server. However, for those times when the application does have to go to the server, speed can still be a problem. We will explore ways to improve interactive speed between the client and the server. One way of doing this is to reduce the time for starting up the server. This can be achieved by running a daemon that provides the services. The CGI script could then be a very simple program that simply opens a socket connection with the daemon and requests it to perform the services on its behalf.

7 Summary

We have taken PARC's WebWriter Editor, an editor for HTML-based applications, and re-designed it to get improved interactive speed, reduced server load, and reduced screen clutter. While the old system used only server-side CGI scripts, the new architecture uses a combination of CGI and client-side JavaScript. The server downloads to the browser a parsed version of the HTML page to be edited; the browser executes JavaScript code to handle editing operations on that page. The JavaScript code itself is structured according to the editor's new frame-based interface, which divides the screen into areas for previewing the page and holding control panels. One or more JavaScript-enhanced HTML modules correspond to each frame, and are downloaded when needed. Updating the screen after a user action often requires only redisplaying a sub-part of the window, such as a frame or a GIF image. The server is only used for fetching and saving files, parsing HTML, and starting up the system.

This architecture, which we call the Meteor Shower Application Architecture, improves interactive performance by generating new pages on the client, thus avoiding network and server-startup delays. The reduced number of accesses to the server also reduce server load. Finally, screen clutter is reduced by using JavaScript image replacement instead of radio buttons to display the current cursor.

Acknowledgments

The authors are grateful for the comments and suggestions of those in the Information Sciences and Technologies Lab at Xerox PARC who tested early versions of the WebWriter II Editor. We also would like to thank the Stanford Digital Library Project and Xerox Corporation for supporting this project.

Author contact information

Arturo Crespo
[http://www.stanford.edu/~crespo]
crespo@cs.stanford.edu
Stanford University
Computer Science Department
Gates Bldg. Office 420
Stanford, CA 94305
Bay-Wei Chang
[http://www.parc.xerox.com/istl/members/bchang/]

bchang@parc.xerox.com
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
Eric A. Bier
[http://www.parc.xerox.com/istl/members/bier/]
bier@parc.xerox.com
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

References

[Apple87] Apple Computer, Inc. HyperCard User's Guide. Apple Computer, Inc., Cupertino, CA, 1987.

[Arnold96] Ken Arnold and James Gosling. The Java Programming Language. Addison-Wesley for Sun Microsystems, 1996.

[Crespo96] Arturo Crespo and Eric A. Bier. "WebWriter: A browser-based editor for constructing web applications." *Fifth International World Wide Web Conference* (Paris, France, May 1996). Computer Networks and ISDN Systems, Vol. 28, 1996, pp. 1291-1306.

[Hudson96] Scott Hudson, Ian Smith. subArctic Home Page. At URL http://www.cc.gatech.edu/gvu/ui/sub_arctic/.

[JavaSoft96a] JavaSoft. The Java Series. At URL http://java.sun.com/Series/.

[JavaSoft96b] JavaSoft. java.awt Documentation. At URL http://java.sun.com/products/JDK/1.0.2/api/Package-java.awt.html.

[Kamba95] Tomonari Kamba, Krishna Bharat, and Michael C. Albers. The Krakatoa Chronicle - An interactive, personalized, newspaper on the web. In Proceedings of the 4th International Conference on the World Wide Web (Boston, December), the World Wide Web Journal, O'Reilly & Associates, Inc. 1995, pages 159-170. At URL http://www.w3.org/pub/Conferences/WWW4/Papers/93/.

[Netscape96a] Netscape Communications Corporation. Netscape JavaScript. At URL http://home.netscape.com/comprod/products/navigator/version_3.0/.

[Netscape96b] Netscape Communications Corporation. PowerStart. At URL http://personal.netscape.com/custom/page/show_page.html.

[Watters96] Aaron Watters, Guido van Rossum, and James Ahlstrom. Internet Programming with Python. MIS Press/Henry Holt Publishers, 1996.