Computer Networks and ISDN Systems 29 (1997) 1507–1517

Responsive interaction for a large Web application: the meteor shower architecture in the WebWriter II Editor

Arturo Crespo[a,*], Bay-Wei Chang[b,1], Eric A. Bier[b,1]

[a]*Computer and Science Department, Stanford University, Gates Bldg. Office 420, Stanford, CA 94305, USA*
[b]*Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA*

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. © 1997 Published by Elsevier Science B.V.

Keywords: Application generators; WWW; Meteor shower; Responsive interaction; Large Web application; WebWriter architecture; Browser-based editor; Server-based WWW applications construction; Web page generating programs; Direct-manipulating Web page editor; Server-based authoring tools

1. Introduction

1.1. The WebWriter application builder

The *WebWriter* system [3] supports the construction of simple interactive web applications without the need to learn HTML or CGI programming. Modeled after HyperCard [1], 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 [8].

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

* Corresponding author. E-mail: crespo@cs.stanford.edu
[1] E-mail: {bchang,bier}@parc.xerox.com

0169-7552/97/$17.00 © 1997 Published by Elsevier Science B.V. All rights reserved.
PII S0169-7552(97)00064-0

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 [5,2]. 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) [6] or the subArctic user interface toolkit [4].

(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 [7]. 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 [9], 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.

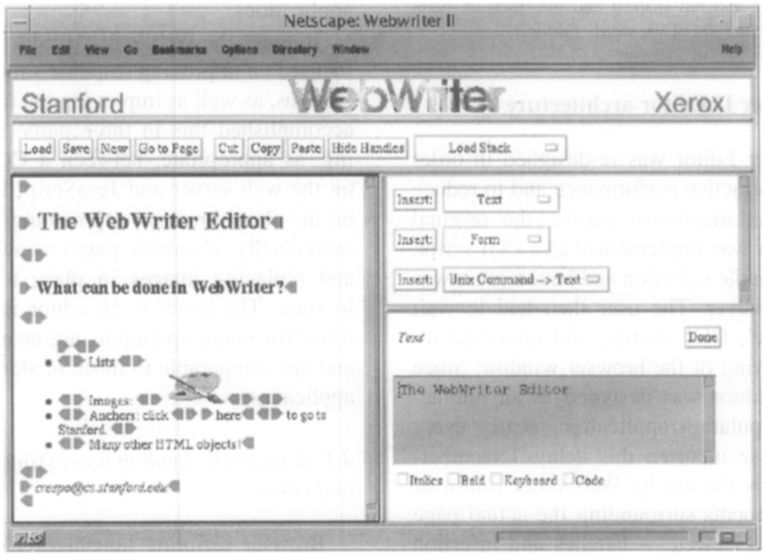
3. The WebWriter II Editor user interface

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

The editor consists of five frames tiling the browser window (see Fig. 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.

In editing mode, the WebWriter II Editor displays the current page as interpreted HTML together with additional images, called *handles*, as shown in Fig. 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.

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 [3].


Netscape: Webwriter II
File Edit View Go Bookmarks Options Directory Window Help
Stanford
WebWriter
Xerox
Load Save New Go to Page Cut Copy Paste Hide Handles
Load Stack
Insert: Text
Insert: Form
Insert: Unix Command -> Text
Text
Done
Italics Bold Keyboard Code

The WebWriter Editor

What can be done in WebWriter?

- Lists
- Images:
- Anchors: click here to go to Stanford.
- Many other HTML objects!

crespo@cs.stanford.edu

The WebWriter Editor

Fig. 1. The WebWriter II Editor.

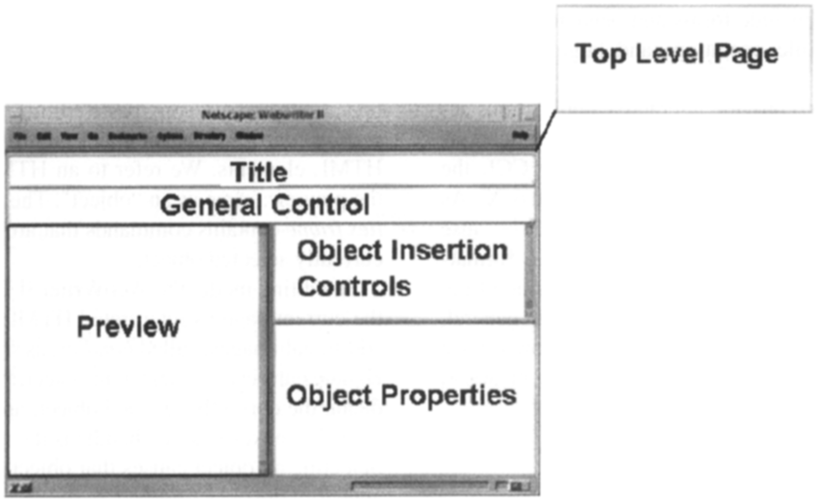


Fig. 2. WebWriter II Editor frames.

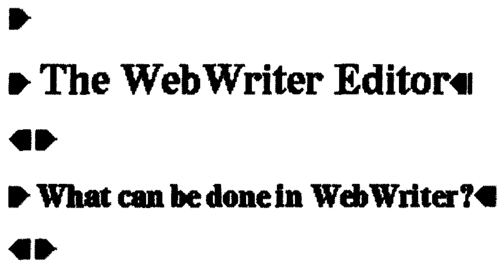


Fig. 3. Handles (grey and red shapes) and the insertion point (vertical black bar to the right of the word "Editor").

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 [10] 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 Fig. 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.

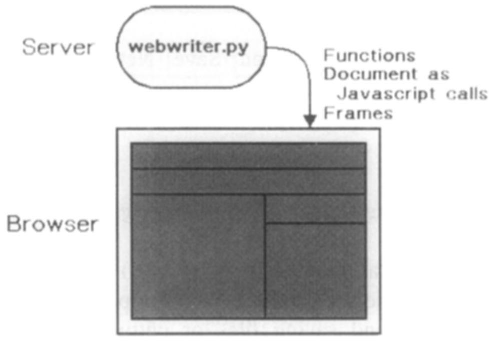


Fig. 4. The server downloads functions, global data structures, and the component frames to the browser.

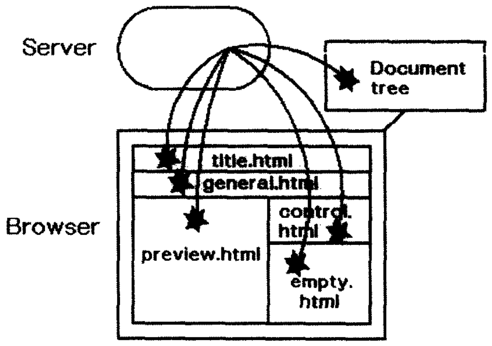


Fig. 5. The server startup meteor shower.

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 Fig. 5.

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.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.