Measuring Client-Perceived Response Times on the WWW

Ramakrishnan Rajamony and Mootaz Elnozahy
IBM Austin Research Lab, 11501 Burnet Road, Austin TX 78758
rajamony@us.ibm.com, mootaz@us.ibm.com

Abstract

The response time of a WWW service often plays an important role in its success or demise. From a user's perspective, the response time is the time elapsed from when a request is initiated at a client to the time that the response is fully loaded by the client. This paper presents a framework for accurately measuring the client-perceived response time in a WWW service. Our framework provides feedback to the service provider and eliminates the uncertainties that are common in existing methods. This feedback can be used to determine whether performance expectations are met, and whether additional resources (e.g. more powerful server or better network connection) are needed. The framework can also be used when a consolidator provides Web hosting service, in which case the framework provides quantitative measures to verify the consolidator's compliance to a specified Service Level Agreement. Our approach assumes the existing infrastructure of the Internet with its current technologies and protocols. No modification is necessary to existing browsers or servers, and we accommodate intermediate proxies that cache documents. The only requirement is to instrument the documents to be measured, which can be done automatically using a tool we provide.

1. Introduction

Businesses increasingly use the World Wide Web (WWW) to supply information such as news and movie reviews, and perform services such as stock and banking transactions for their customers. Responsive service plays a critical role in determining end-user satisfaction. In fact a customer who experiences a large delay after placing a request at a business's web server often switches to a competitor who provides faster service. Two factors contribute to the response time as perceived by a client: the network delay and the server-side latency (the time it takes to generate a response from the time the request reaches the web server). There are many established techniques for reducing the response time over the Web, including the use of powerful servers, reverse proxies at the server, Web caches, and clever load balancing among clustered or geographically dispersed servers. The services that Akamai [1], Digital Island [2], and Inktomi [3] provide are examples of how these techniques could work together.

Given the vital role played by response times in determining end-user satisfaction, businesses need to have quantitative information about the perceived response times of their services. This information guides the reaction of the business if the performance is below expectations. Server latency, i.e., the time it takes to service a request once it enters a web site, is an oft-used metric for infrastructure planning. However, server latency measures do not include network interactions and cannot represent user-perceived response times. For instance, server latency data sheds no light on potential problems within the network such as that caused by a slow Internet connection.

A number of companies today offer to periodically poll Web services using a geographically distributed set of clients. At best, polling can generate an *approximation* of the response time perceived by actual customers. At worst, the geographic and temporal distribution of the polls may be completely different from that of a web site's actual customers, leading to useless response time data. Ultimately, the only reliable way to obtain client-perceived response time is to actually measure it. Examples of such polling services include ServerCheck[4], GoldTest [5], and eValid [6].

This paper presents a novel framework for measuring the *actual* response time perceived by customers as they access a Web service. It does not require a third party, statistical polling, or extra workload. It is applicable to any business, whether they run their services "in-house" or in a consolidated server, and whether the content they serve is statically or dynamically generated. Our framework also works with the existing technology restrictions and limitations. In particular, it does not require any changes to the Hyper Text Transfer Protocol [7], client browsers, or any existing technology. The framework leverages the scripting and event notification mechanisms in HTML 4.01 [8] and uses scripting languages such as JavaScript 1.1 [9] to measure and collect client-perceived response times without any browser modifications. Both HTML 4.0

and JavaScript 1.1 are fairly de facto standards and are supported by the popular Netscape Navigator and Internet Explorer browsers. Finally, our methods do not depend on the use of Java applets [10] and cookies [11], support for which could be disabled by users.

The response time data that we collect can be used in a variety of ways, some of which we describe here. A Web service may use the data to determine whether it is in danger of losing its customers due to intolerable response times. It may also use the data to decide whether a proxy caching service will be useful, and if so, where to place the proxy caches. Web hosting IDCs contract with businesses through Service Level Agreements (SLA), which specify the quality of service that the IDC will provide. The ability to collect accurate client-perceived response time values also enables the creation of a new class of SLAs in which an IDC can contract to serve some fraction of the site's visitors within a specified amount of time. Finally, accurate response time values can be used to differentiate between different proxy caching services.

We begin by providing some background in Section 2 and describe the framework implementation in Section 3. In Section 4, we analyze real response time data we have obtained by instrumenting "The Wondering Minstrels" web site. We describe related work in Section 5 and our conclusions in Section 6.

2. Background and Overview

We begin by defining some basic terms used through the following discussion. A *bundle* is a set of web pages that have been instrumented to measure client-perceived response times. These may be HTML or non-HTML pages, and could be static files or dynamically generated content. The HTML pages in the bundle may contain links to each other, enabling users to traverse the bundle by dereferencing hyperlinks. There can be two kinds of links: *instrumented* and *uninstrumented*. An *instrumented* link points to a page whose response time we want to measure when the link is dereferenced. A page pointed to by an instrumented link is itself instrumented. An *uninstrumented* link points to a page for which we do not want to know the response time.

Users arrive at a specific page within a bundle in one of two ways, causing it to be loaded in their browser. First, the user may have dereferenced an instrumented link from another page within the bundle. We call this an *instrumented* entry into the page. Alternatively, they may have directly entered the page's URL into the browser, or may have followed a link from a page outside the bundle. We term this an *outside* entry into the page.

The framework introduced here allows accurate measurements of all instrumented entries to HTML pages in a bundle. Furthermore, it calculates the response times of each embedded object such as images or Flash animations [12] within a web page.

2.1. HTML Scripts and Event Handlers

The HTML standard specifies that a HTML page can contain embedded client-side *scripts*, which are executed as the page is parsed by the browser[1]. Furthermore, HTML link elements can contain a script snippet instead of a URI. When a link containing a script snippet is dereferenced, the script is executed. The following paraphrases the HTML standard's script description [8].

> A client-side script is a program that may accompany an HTML document or be embedded directly in it. The program executes on the client's machine when the document loads, or at some other time such as when a link is activated. Authors may attach a script to a HTML document such that it gets executed every time a specific event occurs. Such scripts may be assigned to a number of elements via the intrinsic event attributes. Script support in HTML is independent of the scripting language.

Browsers today support a wide variety of scripting languages. JavaScript [9], VBScript [13], and Tcl [14] are three examples of popular scripting languages. JavaScript is by far the most popular scripting language, and is supported by virtually all browsers that allow scripting [15].

The HTML standard also specifies several intrinsic *events* and the interfaces through which a client-side scripts can be invoked when different events occur. The Timekeeper uses three specific events that can invoke a script attached to a document:

- onclick(): In the context of a link, the onclick handler is invoked when the user clicks on a link. The link is de-referenced based on the handler's return value.
- onload(): In the context of a document, the onload handler is triggered when a document and its embedded elements have been fully loaded. In the

[1] A browser can defer script parsing only if the "defer" attribute of the SCRIPT tag is set true. In it absence, scripts must be parsed as they are encountered.

context of an OBJECT element, the onload handler is triggered when the object is fully loaded.

- onunload(): Triggered when a document is about to be unloaded to make room for a new document, or when the browser is being closed.

2.1. Overview

Abstractly, our scheme consists of the following four basic steps. We describe the details of each step later in the paper.

1. Before sending a request to a web server, the client browser is made to determine and remember the current time. This action is performed using a client-side script embedded within the currently displayed page.
2. The normal browser actions cause the requested page and its embedded elements to be loaded and displayed.
3. After the response is fully received, the client browser computes the response time as the difference between the current time and the start time remembered from step 1. We again use a client-side script to carry out the computation, initiating this step through an onload event handler that triggers when the page fully loads. Note that since both the start and end times are computed on the client, no clock synchronization is required.
4. Once a response time value is determined, client-side script action transmits it to a "record keeper" web site based on an established policy. Several policies may be implemented. For instance, response time samples could be submitted when they are collected, submitted in bulk, or submitted only when they are above a particular threshold value. The record keeper could be any web server on the WWW, enabling an agent other than the origin web server to collect and maintain response time values.

Steps 1, 3, and 4 are divided between two agents within the browser called the *Timekeeper* and the *Librarian*:

- The *Timekeeper* is responsible for computing the response time values. In doing so, it uses the *Librarian* to save and retrieve state values. Depending on a prespecified policy, the Timekeeper also communicates the computed response time samples along with other information to a record keeping web site.
- The *Librarian* stores and retrieves state values upon request. It provides the Timekeeper with a well-defined interface for performing these actions.

The following sections discuss these mechanisms in detail. We use JavaScript as the scripting language in our prototype implementation. Originally introduced by Netscape, JavaScript has become the de facto scripting language on the WWW, supported by most popular browsers including Navigator and Explorer. However, our approach is equally valid with other scripting languages such as VBScript[13] or Tcl [14].

3. Implementation

3.1. Timekeeper

The Timekeeper determines the current time and computes time deltas. It occasionally uses the Librarian to check in new time samples and to check them out later. It carries out these actions using HTML events and scripts. The Timekeeper can also be made to record the identity of each instrumented page or image along with the corresponding response time sample.

Several policies may be implemented. For instance, response time samples could be submitted when they are collected, submitted in bulk, or submitted only when they are above a particular threshold value. Under some conditions, the record keeper can be any computer on the WWW. This could serve having a third party verify Service Level Agreements.

Consider an instrumented link in a web page. By definition, the link points to a page that is itself instrumented. When a user clicks on the link, a Timekeeper script is invoked. The script determines the current time, records it using the Librarian, and then permits the link to be dereferenced. After this page is fully loaded, it's onload handler is invoked. The handler obtains the time at which the page request was made by querying the Librarian. It then calculates the response time as the difference between the current time and the "send-time". If the measured response time is to be transmitted later, the handler records it using the Librarian. Figure 1 summarizes these actions.

When the user directly types the location of a document in the address bar of a browser, the currently displayed document in the browser window is replaced with the one requested by the user. Before loading the new document, the browser invokes an onunload handler, if one has been specified. The Timekeeper performs cleanup operations on this event, communicating the collected response time values to a record keeper. A cleanup operation is also initiated when the user closes the browser window.

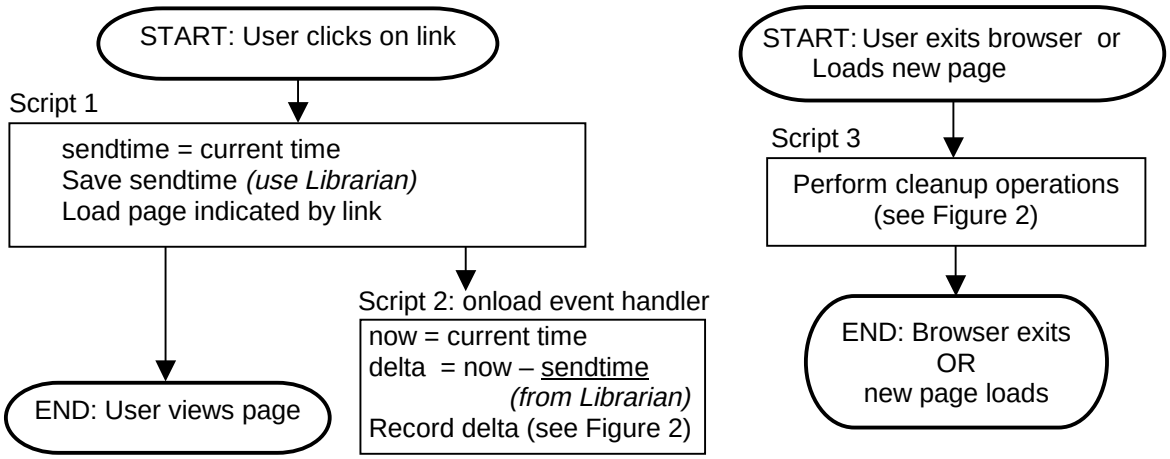

Figure 1: Timekeeper actions

The cleanup operations performed by the Timekeeper and its actions for informing the record keeper are summarized in Figure 2. There are two points of entry shown in the figure, corresponding to the unexplained actions in Figure 1. The left hand side depicts the Timekeeper actions on individual response time samples. The right hand side shows the Timekeeper actions if the browser window is closed or if the user directly loads a new page.

The Timekeeper may implement any desired policy to communicate the response time samples. For instance, in order to amortize the cost of communicating the response time samples to the record keeper, a policy to collect some number of samples before communicating them could be implemented. Alternately, the policy may be to communicate only response times greater than an upper bound. The Timekeeper actions in Figure 2 implement the former ("communicate samples in bulk") policy. The next section describes the Timekeeper actions to communicate the collected response time samples in greater detail.

3.2 Transmitting Response Time Values

The response time samples collected by the Timekeeper and saved by the Librarian must be communicated to a *record keeper* in order to be of use. Our framework permits the record keeper can be the source web server itself, or any other web server on the Internet. Several policies could be established for communicating the samples. Some examples are:

- Communicate on every sample
- Communicate after accumulating a prescribed number of samples

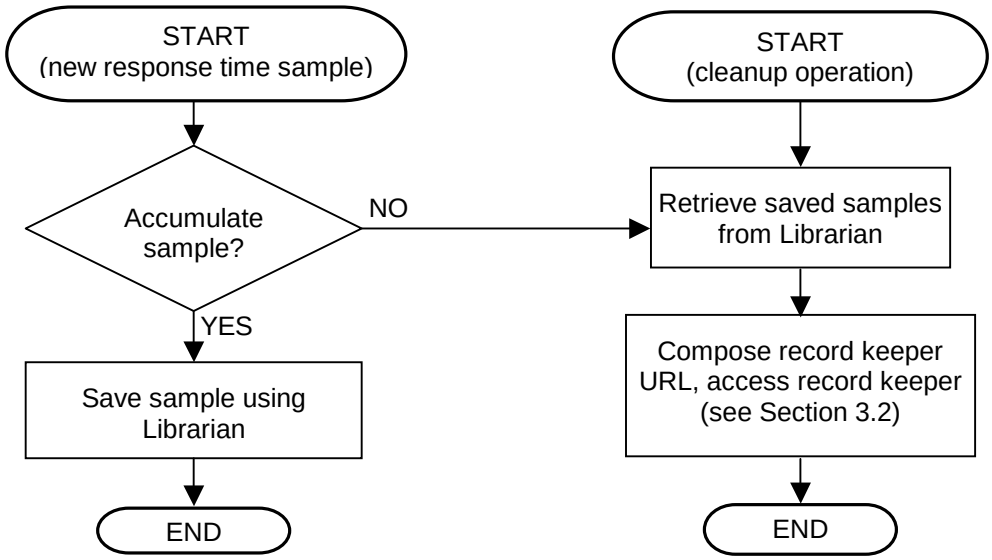

Figure 2: Timekeeper operations contd. (implements "communicate samples in bulk" policy)

- Communicate only those samples that exceed a pre-established threshold value.
- Communicate only samples from favored customers (as determined by a cookie).

For any established policy, we assume that the source web server provides the Timekeeper with a JavaScript function to determine when and what samples to communicate.

The Timekeeper uses a JavaScript Image object in order to perform the communication. Image objects were introduced in client—side JavaScript 1.1 and are primarily used to preload images [9]. By setting the src attribute of an image object to a desired URL, the specified content is loaded and placed in the browser's cache. By initiating actions such as image replacement and animation only after all the desired images are preloaded, the quality of visual effects in the browser can be enhanced. Image loading does not block JavaScript execution. Instead, the image content is loaded asynchronously alongside other browser actions.

When the Timekeeper needs to communicate with the record keeper, it composes a special URL containing the data. This URL accesses a script on the record keeper site. The record keeper extracts information from the URL, and replies with a response. The response is set to be non-cacheable in order to prevent intervening proxies from squelching record keeper communication.

HTML specifications stipulate that if the currently loaded document has an unload event handler specified, that handler must be invoked before a new document is loaded. The unload handler is also invoked if the user closes the browser window. The Timekeeper uses this event handler to communicate with the record keeper when the user visits an uninstrumented page or makes an outside entry to a page in the bundle.

Using the image preload technique to communicate with the record keeper on a browser close is problematic. Consider a case where the user has closed the browser, and the unload handler has initiated the record keeper image load. Since image loading is done asynchronously, the browser may die even before the TCP connection to the record keeper is fully open. Unfortunately, to the best of our knowledge, neither Navigator nor Explorer support UDP URL schemes. Consequently, we use a different method for record keeper communication on a browser close.

In both Navigator and Explorer, a new browser window can be reliably opened from an unload handler. A new window is opened by using the open method of the current window object[2]. We use this feature to communicate with the record keeper by opening a new window with the response time URL. The record keeper is set up to respond with a page that closes the window. This is done with a page with an onload handler that performs a "window.close( )" operation.

Since the unload handler is invoked when each document is displaced from the browser window, the Timekeeper needs to distinguish between the unload handler invocations on an instrumented entry to the next page, or on an outside entry to an arbitrary page that may be inside or outside the bundle. We make the distinction by setting a variable in the window object before carrying out an instrumented entry to the next page. The unload handler checks whether this variable is set in the window object. If it finds the variable set, we know that the current page is being displaced to make room for an instrumented page in the bundle. Otherwise, the Timekeeper performs the cleanup operations from within the unload handler. We reset this variable in readiness for the next unload within the onload handler that is invoked after an instrumented page has fully loaded.

Since opening a separate communication window could detract from the end-user experience, the Timekeeper has two choices. First, it can make the communication window small (but see Section 3.3.2). Second, it can decide that losing the last few collected samples is better than opening a new window.

Yet another option is to use a cookie. If the client browser has cookies enabled, the Timekeeper could also save the information in a cookie for transmission at a later date. This cookie could be expired as soon as a request carrying it is sent to the source web server, causing it to no longer be sent out on future requests. Due to JavaScript security restrictions that control access to cookies from scripts, the record keeper will have to be the same as the source web server in this case.

3.3. The Librarian

The Librarian is responsible for storing and retrieving time samples on behalf of the Timekeeper. At present, to the best of our knowledge, there is no straightforward mechanism in a browser to maintain state across page loads. In particular, for security reasons, browsers do not permit client-side scripts to maintain state across

[2] For example, enable JavaScript in your browser and visit http://www.cs.rice.edu/~rrk/neverclose.html. Many web sites use this annoying technique to make it difficult for users to leave their site.

```
function OpenStateWindow( )
{ // Open a state window if needed
   var h = self.open (“”, “statewin”,
            “width=100,height=100,location=no”);
   if (typeof h.valid == “undefined”) {
      h.document.write (“Benign msg for user”);
      h.document.close ( );
      h.valid = true;  // Don’t write next time
    }
    return h;
}

function SaveSendtime ( )  {
   var h = OpenStateWindow ( );
    h.sendtime = (new Date( )).getTime ( );
}

function GetSendtime ( )  {
    var h = OpenStateWindow ( );
    return h.sendtime;    // can be undefined
}

function RTonload ( )  {  // onload handler
   var sendtime = GetSendtime ( );
    if (typeof sendtime != “undefined”)  {
       var now = new Date ( );
       var delta = now.getTime ( ) – sendtime;
       // Accumulate delta, or transmit now
    }
}

function RTonclick ( )  {  // link onclick handler
    SaveSendtime ( );
    return true;  // Permit link to be dereferenced
}
```

Figure 3: Timekeeper operation with Separate Window Librarian

page loads. When a new document is loaded, all of the scripts and variables associated with a page are cleared. In this section, we present three approaches for implementing the Librarian without resorting to browser modifications.

3.3.1 Saving State in a Cookie

HTTP, the protocol used for retrieving web pages, is inherently stateless [7]. Cookies were introduced as a mechanism to enable clients to build stateful sessions on top of HTTP [11]. Simply stated, a cookie is a tag created by the server and delivered to the client along with an HTTP response. On subsequent requests to the same server, the client presents the tag along with its request. Cookies permit a session to be built using individual HTTP transactions.

Cookies can be subverted for maintaining state. While it is the server that typically sets cookies, client-side scripts also have the ability to set, modify, and retrieve cookies. Thus, the Librarian could use cookies to maintain state across page loads. The Librarian could use JavaScript when state needs to be saved, and retrieve it again using JavaScript.

The naïve approach described above has a severe drawback. Today, the WWW is dotted with caches and proxies. The idea behind caching is to place an object “closer” to the user, reducing the demands placed by a request on both the network and the origin web server, enabling it to be serviced quickly. However, since cookies are used to create sessions out of HTTP, a cache that observes a request with an attached cookie typically does *not* service the request forwarding it instead to the origin web server [7]. This is the correct approach since two requests for the same URL with different cookies could potentially lead to different responses. Consequently, using cookies to maintain state would cause each request to be sent to the origin web server, negating the usefulness of proxies and caches. The cookie approach is therefore practical only when the content being delivered is itself dynamic, with no intervening proxy caching it.

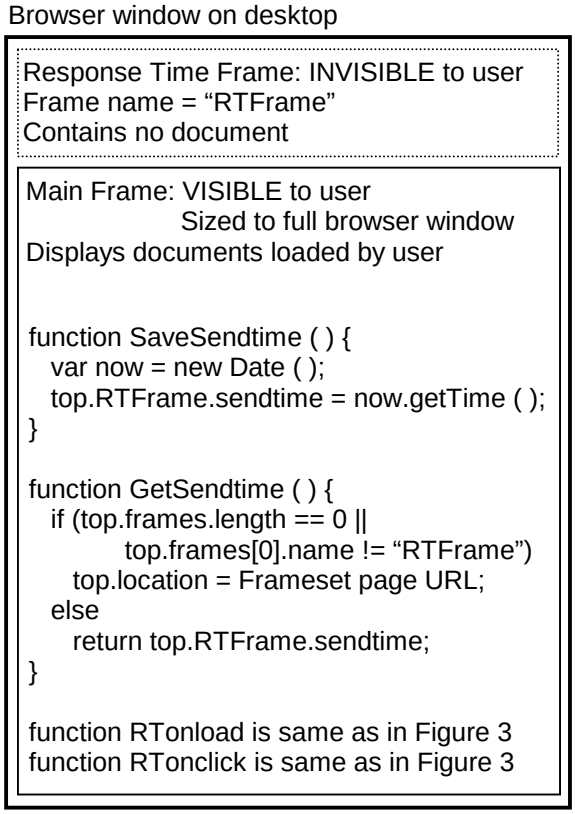

Figure 4: Timekeeper and Librarian operations when using frames

3.3.2 Using a Separate Window

Since client-side JavaScript enables state to be stored in a window object, the simplest approach is to open a new window on the first outside entry to a web page in the bundle and to save the state there. For as long as this window stays open and until a different URL is loaded in it, state stored in its context can be recovered. Figure 3 shows sample JavaScript code that achieves this goal.

One limitation of this approach is the presence of the state window on the user's desktop. Even though this window is small and can be made to contain a benign message, a user may arbitrarily close the window. Furthermore, JavaScript security restrictions prevent a script from opening a window in a minimized state, or with a size smaller than a prescribed minimum, unless the script has the UniversalBrowserWrite privilege. While this privilege can be obtained by involving the user, the process is fairly awkward and cumbersome.

3.3.3 The Frame Approach

In the previous section we described how a separate browser window on the user's desktop could be used to save state. However, in client-side JavaScript, the *window object* does not have a one-to-one correlation with a browser window. More specifically, each HTML *frame* within a browser window corresponds to a separate window object. This correspondence paves the way for us to save state in the window object context of a frame *within* the same browser window as that displaying the document being loaded by the user.

The window object is the global object and execution context in client-side JavaScript. There is no direct correlation between a *window* as viewed in a desktop environment, and the *window object*. A window object is created for each desktop-level window *or* frame within a browser desktop window that displays a HTML document. From our perspective, the interesting property of client-side JavaScript is that it permits scripts executing in the context of one window object to access variables and scripts executing in another window object's context [9].

The HTML standard describes frames as follows, permitting frames to be created with zero size [8]. Such frames will be invisible to the user.

> HTML frames allow authors to present documents in multiple views, which may be independent windows or subwindows. Multiple views offer designers a way to keep certain information visible, while other views are scrolled or replaced. For example, within the same window, one frame might display a static banner, a second a navigation menu, and a third the main document that can be scrolled through or replaced by navigating in the second frame.

The Librarian can use frames to save state by placing each instrumented page in the bundle within a frameset document that divides the top-level browser window into two frames: the response-time frame and the main frame. The response-time frame is set to zero size and is therefore not visible to the user. The main frame holds the instrumented content.

There are two ways a user can make an uninstrumented entry to a page. First, a user could directly enter the URL of the frameset document. This causes both the response-time frame and the data frame to be loaded within the browser. Alternately, the user could directly enter the URL of the data frame in the browser's location bar. To force the browser to load the corresponding frameset document, each instrumented page in the bundle contains JavaScript to check for the existence of the response-time frame. The existence check is made after the main frame has loaded. If the response time frame does not exist, the JavaScript forces the corresponding frameset document to be loaded in the top-level browser window.

State saving is accomplished exactly as in the case with the separate window. Figure 4 explains the approach, showing how the Timekeeper and the Librarian interact together to determine the response time values.

Visiting the site through a browser window causes the following actions to take place. On the first outside entry to an uninstrumented page, the JavaScript actions in Figure 4 cause the corresponding frameset document to be loaded. As long as the user makes instrumented entries to the other pages in the bundle, the response time frame stays in the top-level browser window. The response-time frame needs to be reloaded only if the user makes an uninstrumented entry to a page in the bundle.

The main limitation of this approach is the need for loading the frameset document on uninstrumented entries to pages in the bundle. When encountering a page with frames, the browser first obtains the "container" frameset document. Only after receiving the container frameset can the browser determine what documents to obtain and render in the internal frames. This could give rise to extra client-server transactions and delays for the user.

Three factors mitigate the problem caused by the frameset document. First, the web site might already have a frame that exists on all pages, enabling the Librarian to simply use that frame. Second, the extra client-server transactions (when the Librarian uses a dedicated frame) are encountered only during the *initial*

uninstrumented entry to a web page. Subsequent instrumented browsing occurs at full speed. Finally, it may be possible to avoid the extra transactions by setting a long lifetime for the frameset documents. HTTP permits content to be delivered with explicit expiration times, allowing intervening caches and the client to cache content and use it *without checking for validity* against the origin web server. By providing the frameset document with a long lifetime, the browser needs to fetch the container frameset only when it is absent from the local browser cache. Consequently, when the user revisits a bundle at periodic intervals, as often happens since people are creatures of habit, the browser will be able to use a cached copy of the frameset document. Only the main frame's content will need to be fetched in such cases.

3.3.4 Using the Window Name

In Section 3.3.3 we described the window object in client-side JavaScript. Every window object has a name property. This property exists primarily for use as the value of a HTML TARGET attribute in the <A> or <FORM> tags. In essence, the TARGET attribute enables an anchor or form to display its results (when the linked document is dereferenced or the form is submitted) in the window with the supplied name.

The initial window and all new browser windows opened by most versions of both Explorer and Navigator have no pre-defined name property. Consequently, these windows cannot be addressed with a TARGET attribute. The name property is read-only in JavaScript 1.0, creating a problem when the initial window has to be addressed. JavaScript 1.1 resolves this problem by enabling the name attribute to be modified from within a script [9].

When a new page is loaded in a window, all of the scripts and variables associated with the window object are cleared. However, a window's name property *persists* across page loads. Thus a web page loaded into a window can be the target of actions in another window, even if the loaded content does not explicitly set the window name. This feature enables the same content to be loaded in the main browser window or in a popup window, depending on the context from where it is referenced.

We can leverage the persistence of the name property to store state. The idea is to append the desired state to the window name and to retrieve it from there, restoring the name after the retrieval[3]. Figure 5 illustrates the details of this approach.

```
function SaveSendtime ( )  {
   var s = (new Date( )).getTime ( );
   window.name += '_RT_' + s;
}

function GetSendtime ( ) {
   var n = window.name;
   var rt = n.indexOf ('_RT_');
   var s = n.substring (rt+4, n.length);
   window.name = n.substring (0, rt);
   return s;
}

function RTonload is same as in Figure 3
function RTonclick is same as in Figure 3
```

Figure 5: Timekeeper and Librarian operations when using window names

An obvious limitation of using the window name is the race condition it introduces. In order to compute response times, the window name is changed just prior to a new document being loaded in a window. The saved state is retrieved and the window name restored only when the document has fully loaded. During this interval, the window cannot be referred to by its original name.

The race condition causes a problem partly due to the awkward interface that client-side JavaScript provides for referring to a window. Given a window name, the only way to obtain a reference to the window is by using the window.open method. The name supplied to the method must be exact, with no wildcards allowed. Furthermore, if a window with the supplied name does not exist, the method simply opens a new window with that name. These properties of the open interface imply that if a window with an ongoing instrumented page load is targeted, a new window with the supplied name is opened. This may distract and confuse the user.

In the next section, we describe how the window name approach can be combined with the use of separate windows to yield a practical, useful solution.

3.3.5. Separate Windows + Window Names

Taken individually, the separate window and window name schemes suffer from limitations when used to implement the Librarian. The separate window could be a distraction for the user, since it must remain open for

[3] We have empirically determined that both Navigator and Explorer support names over 1000 characters.

as long as there are instrumented entries to the bundle. The window name scheme exposes a race condition that could open a new browser window on the user's desktop, contrary to the content developer's intent.

However, the separate window and the window name schemes can be combined together to yield a practical solution that does not suffer from these limitations. The idea is to divide the windows in which the instrumented pages will be displayed into two sets: the *main* windows and the *child* windows. Main windows are those that are not targets for any content. Child windows are those in which content is loaded by actions in both main and child windows.

For a page that is loaded in a main window, the Librarian saves state in that window's name. For a page that is loaded in a child window, the Librarian saves state as a property in the context of the child window's ancestor. The ancestor can be determined by following the opener property of the child window, which is a reference to the window object that opened the child window. State saving and retrieving will be accomplished by using a combination of the scripts shown in Figures 4 and 5. Note that it is totally safe to change a main window's name, since by definition, a main window can never be the target for any content.

3.4. Limitations

Our schemes have the following limitations:

1. We cannot compute response times for outside entries to instrumented web pages. For example, we cannot compute response times for pages loaded by directly entering a URL into a browser's location bar.

2. Our scheme handles instrumented entries to HTML pages and objects embedded within HTML pages. We cannot compute response times for other MIME types. For instance, we cannot compute the response times for pure images (outside of a HTML document) directly loaded by a browser, or for PDF documents loaded and displayed by the Adobe plugin. In order to handle such MIME types, we need plugin support in the form of a public method that can be used to determine the load status of the content. Flash animations are an example of a MIME type we can handle even when the animation is loaded outside of an <OBJECT> context, by using the PercentLoaded public method of the MacroMedia Flash plugin [16].

3. Our implementation uses client-side JavaScript 1.1. Consequently, it will not work with browsers that do not support JavaScript 1.1, or that have disabled JavaScript. In particular, our implementation requires Navigator versions above 3.x, and Explorer versions above 4.0.

4. Status and Evaluation

We have implemented our scheme on "The Wondering Minstrels," a poem-of-the-day web site that receives several hundred hits a day. The site can be accessed at http://www.cs.rice.edu/~rrk/minstrels.html, and has over 650 poems (now) that range in size from 2.6KBytes to 30 Kbytes. A few files that index all the poems take up about 230Kbytes. We instrumented this site using an automated tool we have developed, setting the record keeper to be www.cs.utexas.edu. Note that the origin web server and the record keeper are different servers.

	Any time	00:00 to 07:59	08:00 to 15:59	16:00 to 23:59
All	971	161	411	399
Unknown TLD	92	25	44	23
TLD: .us, .com, .org, .net, .edu, .ca	685	72	285	328
All other TLDs	194	64	82	48

Table 1: Distribution of collected response time values

Table 1 shows the distribution of the response time values we have collected from real visitors to the site for files between 4400 and 4900 bytes in size. As of Jan 21, 2001, these were the most frequently accessed files from the site. The actual web server response is larger by about 260 bytes, which is the approximate size of the HTTP header for these responses. We have broken down the collected values by the time when the request was serviced, and by the top-level domain (TLD) of the client making the request. All times listed are behind UTC by 6 hours. We were unable to resolve hostnames for a number of clients accessing the site. These unresolved hostnames are listed as "Unknown" and account for 9.5% of the collected values.

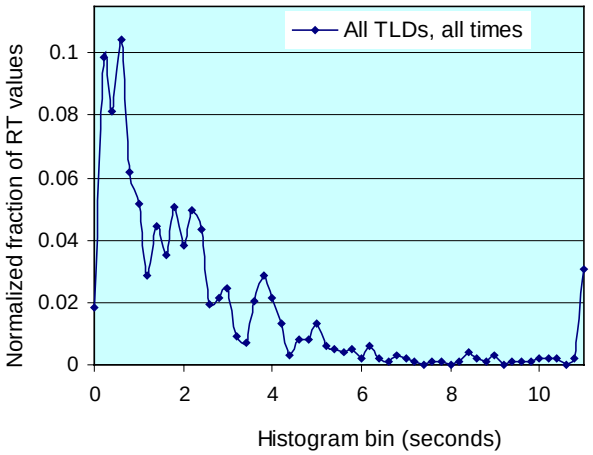

Figure 6: Distribution of collected response time values

Figure 6 shows a histogram of the normalized response time values, irrespective of the TLD from where or the time the request was made. Each data point in the figure indicates the number of response time values that fall into a 200 ms bin to the right of that point. All response times exceeding 10.8 seconds are aggregated and placed into the bin at 11 seconds. There are several interesting points to note about the figure. First, a small fraction of the requests (about 1.8%) were satisfied within 200ms. This is probably due to a cache hit in the client itself, or in a proxy very close to it. Second, by considering the average attention span window to be 4 seconds (i.e., "I want my page to load in 4 seconds or I get bored"), we see that 16% of the responses were served outside the average attention span window.

Breaking down the response time values by time period illustrates the variation in response times over the course of a day. Figures 7, 8, and 9 show the response time values for the midnight–8AM, 8AM–4PM, and 4PM-Midnight time windows respectively. These figures show that the fraction of pages that take longer than 4 seconds to load decreases from 22% in the night, to 19% during the morning, and to 11% during the afternoon.

Figures 10 and 11 break up the response time values by top-level domain. We group together all com, org, edu, net, us, and ca domains into one group, and all other known TLDs into another. We realize that com, org, and net clients could be spread throughout the world. However, this crude division allows us to focus on the response times faced by visitors located far away from the Minstrels web site.

Figure 11 clearly shows the larger response times seen by far-away visitors. About 26% of the users in this group faced delays of more than 4 seconds. Further analysis reveals that the two largest set of visitors in this

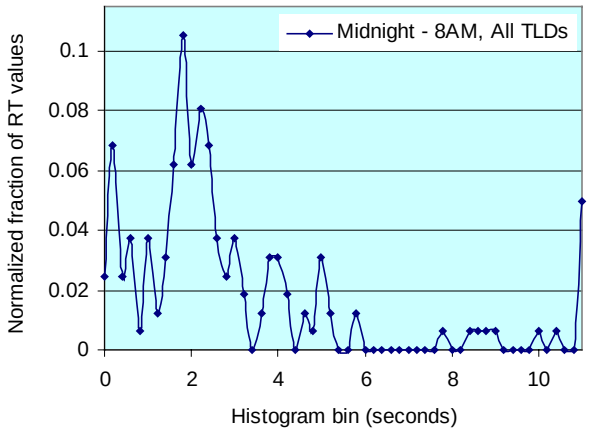

Figure 7: Response time distribution from 0000-0759 CST

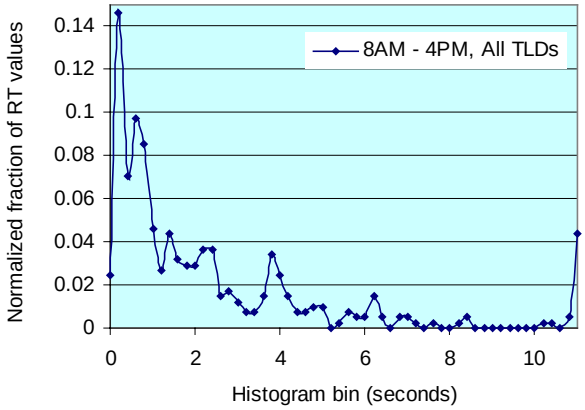

Figure 8: Response time distribution from 0800-1559 CST

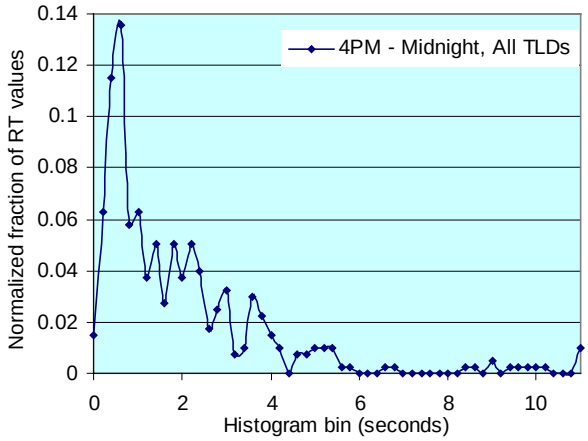

Figure 9: Response time distribution from 1600-2359 CST

group are from the United Kingdom and Argentina. While only 12.5% of the English faced response times larger than 4 seconds, fully 58% of the Argentines fell into the same category. Interestingly, if the Argentine visitors were all served within 4 seconds, only 15% of the far-away visitors would have waited for more than 4 seconds. These results indicate that if the Minstrels site

were to contract with a proxy caching service, placing just one cache in close proximity to the Argentine visitors would provide the most benefit.

The nature of analysis presented here is similar to what a commercial site would want to conduct on its visitors. Our framework enables such an analysis and shows how a site appears to different user groups accessing it. We would like to stress here that such an analysis would not be possible using information gathered at the server only, as it would lack the actual networking effects. Also, we would like to stress that traditional approaches such as pinging the server would have a prohibitive cost emulating all possible users and their geographic distribution. Our experience with measuring actual response times seen by real visitors to the Minstrels site illustrates the need for a measurement framework such as ours. The analysis we describe here is one of many that can be carried out with real response-time data.

5. Related Work

A number of web sites contract with third party companies to poll their servers at periodic intervals. Typically, a battery of geographically distributed clients "ping" the server with fictitious transactions. The site owner specifies the frequency of polling and the geographic distribution. Examples of such polling services include ServerCheck™ [4], GoldTest™ [5], and eValid™ [6]. Polling suffers from several drawbacks. First, the data obtained through polling is a statistical approximation to the response time seen by real visitors to the web site. Polling can also increase the load generated on a site's servers. Ensuring accurate or complete geographic coverage using polling is also difficult. Finally, some services such as financial transactions may be cumbersome to measure using fictitious requests.

Candle's eBusiness Assurance (eBA™) product [17] provides an accurate breakdown of the time spent by a user on a web page. eBA uses an applet to store state on the client browser between page loads. The time at which the user clicks on a link is stored in the applet's context and the response time computed after the page fully loads within the browser. eBA appears to have been introduced around September 2000. In comparison to our work, eBA's main difference is its dependence on a Java applet in order to save state. Since applets are still not widely used owing to their (as perceived) heavyweight nature, this is a fairly significant drawback. In contrast, our scheme requires only JavaScript support in order to compute response times. A significant fraction of the sites on the web today (IBM, CNN, ETrade, CNBC, Disney, etc.) use JavaScript, greatly reducing the barrier to using a scheme such as ours.

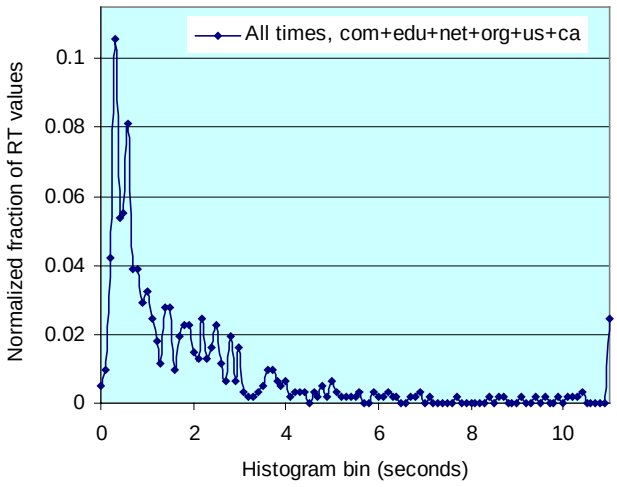

Figure 10: Response time distribution for shown TLDs

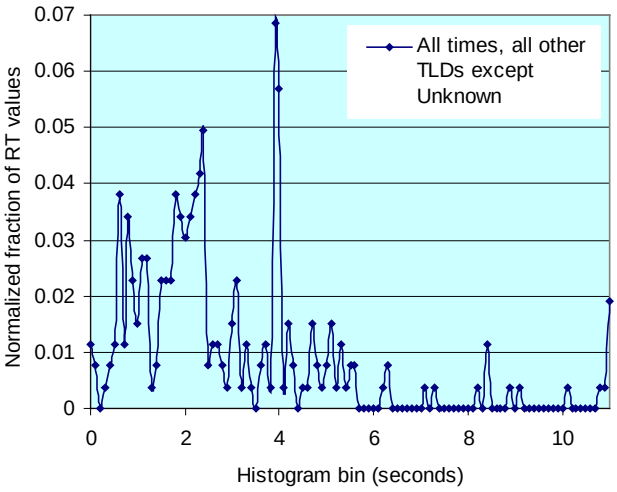

Figure 11: Response time distribution of "far away" TLDs

Tivoli has also introduced a product around September 2000 called Quality of Service Monitor (QoS), as part of their Web Management Solutions package [18]. QoS uses a proxy that sits inside the web site firewall to intercept responses as they leave the site. The proxy adds a small amount of JavaScript code to the content and also remembers the time at which the response is sent back. The added script contacts the QoS proxy when the page has finished rendering. QoS then approximates the response time seen by the client as the delta between when the response is sent to the client, and when the proxy hears back from the client. There are two main differences between the QoS work and ours. First, in order to avoid clock synchronization problems, the record keeper must be the QoS proxy. Second, QoS calculates the response time from the point of view of the proxy, giving rise to three important implications. First, response times can be computed

only if the client contacts the proxy after each page is rendered, adding an extra client--proxy communication to every transaction. Second, the client most likely does not have to create a new TCP connection to communicate with the proxy after rendering the page. Consequently, the measured response time does not take TCP connection setup times into account. Third, QoS cannot determine response times for requests fulfilled by intermediate proxy caches. QoS is more suited for environments such as e-commerce transactions, where the origin server is likely to be involved in every response. Furthermore, QoS also measures the back-end transaction service time (by tagging requests as the enter the site), and the page render time (measured by the added script), both of which are very valuable.

6. Conclusions

On the World Wide Web, the response time seen by a client is a key metric that determines end-user satisfaction. Most schemes in existence today rely on polling the web server using a set of geographically dispersed clients in order to obtain a representative set of response time samples. Polling yields data that is at worst inaccurate, and at best, statistical in nature.

In this paper, we have presented a framework for accurately measuring the response time perceived by a client browser. We are able to measure response times for all visits to pages instrumented using our framework, through hyperlinks that have been instrumented. We can also measure response times for all objects embedded within a web page. Our framework imposes very little overhead on the client computer, and fits the needs of various *existing* commercial web sites.

We divide the work in collecting response time samples between a *Timekeeper*, who computes all time values, and a *Librarian*, who provides the Timekeeper with services for saving and retrieving state. The Timekeeper implementation is fairly straightforward. On the other hand, implementing the Librarian is challenging owing to the difficulty in persisting state across page loads in existing browsers. We present several solutions to this problem, which are well suited for use in existing commercial web sites.

Acknowledgements

We would like to thank Sitaram Iyer for allowing us to instrument the Wondering Minstrels web site and allowing us to publish the aggregate data. We would also like to thank the Computer Science departments at Rice University and UT Austin for use of their web servers. Brian Pinkerton, our shepherd, made many valuable comments that have improved the quality of this paper. Finally, we would like to thank our anonymous reviewers for their helpful insights.

References

1. Akamai FreeFlow™, http://www.akamai.com/html/en/sv/content_delivery.html, Akamai Technologies Inc.
2. Digital Island Inc., Footprint™, http://www.digitalisland.net/services/cd/footprint.shtml
3. Inktomi Corporation, www.inktomi.com
4. NetMechanics ServerCheck™: http://www.netmechanics.com/monitor.htm, NetMechanics Inc.
5. GoldTest™: http://www.keynote.com/services/services/keyreadiness/goldtest.html, Keynote Systems Inc.
6. eValid™ Test Services: http://www.soft.com/eValid/Services/Monitoring/index.html, Software Research Inc.
7. J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol – HTTP 1.1", RFC 2616, June 1999.
8. Dave Raggett, Arnaud Le Hors, and Ian Jacobs, "HTML 4.01 Specification", W3C, December 1999.
9. David Flanagan. JavaScript, The Definitive Guide, O'Reilly & Associates Inc., 1998
10. K. Arnold, J. Gosling, and D. Holmes, "The Java Programming Language", Addison-Wesley, 2000.
11. D. Kristol and L Montulli, "HTTP State Management Mechanism", RFC 2109, February 1997.
12. Macromedia Flash 5, http://www.macromedia.com/software/flash/, Macromedia Inc.
13. Microsoft Scripting Technologies: VBScript, http://msdn.microsoft.com/scripting/default.htm?/scripting/vbscript/default.htm, Microsoft Corporation.
14. John K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley Publishing Company, 1994.
15. Microsoft Corporation, "Using VBScript and Jscript on a web page", http://msdn.microsoft.com/library/techart/msdn_vbnjscrpt.htm
16. Flash methods, http://www.macromedia.com/support/flash/publishexport/scriptingwithflash/scriptingwithflash_03.html, Macromedia Inc.
17. eBusiness Assurance™, http://www.candle.com/solutions_t/enduser_solutions/site_performance_analysis_external/index.html, Candle Corporation.
18. Tivoli Web Management Solutions, http://www.tivoli.com/products/demos/twsm.html, IBM Corporation