WEBARM: Mobile Code Based Agent for Web Application Response Measurement – Software Implementations and Analysis

Joseph V. Elarde and Gregory B. Brewster

DePaul University
jelarde@cs.depaul.edu

Abstract. WEBARM enables end-to-end response time measurement for Web applications through Web page instrumentation. WEBARM-agents are a form of mobile code capable of moving with the Web application to monitor response time. WEBARM includes the software components responsible for interception and processing of the WEBARM API calls including support for response time measurement, collection, and communication. This work provides an assessment of WEBARM instrumentation impact on application response times as well as insights into the design issues involved. We first provide a general overview of alternative approaches to Web response time measurement. We then consider alternative WEBARM agent software designs, focusing on techniques to store timers across Web page references and communicate logged data to a server. Next, we present measurement results for the implementation alternatives to provide insight into the overhead involved with instrumentation and the developed designs. Finally, we present conclusions and a description of future work.

1 Introduction

The dramatic expansion in use of the Internet represents one of the major technological revolutions of recent history [2]. As organizations deploy Web applications over the Internet, we have entered an era where direct customer contact with an organization's Web applications is commonplace. The quality of this experience in terms of functionality, performance, and availability will ultimately affect the customer's view of the organization. Thus, maintaining a high performance Web application is directly related to customer satisfaction [1] and ultimately an organization's success.

The solution for any performance issue begins with a program of objective measurements - knowing what service is delivered can lead to the necessary root cause identification and corrective action implementation processes. Not knowing the service delivered to the end-customer is dangerous - equivalent to not knowing an organization's financial results. So given that service (response time) measurement is important - how should a comprehensive Web response-time measurement program

be implemented? The alternatives we consider in this paper are server-based estimation, site monitoring, and client agent based measurement.

- **Server-based measurement:** Web Server logs or network monitors local to the Web server are used to estimate end-to-end response time delivered to the end-customer (Passive Monitoring).
- **Site monitoring:** Utilizes strategically deployed workstations called "probes" to execute scripted transactions at various locations within the network. The probe machines capture and analyze the scripted transaction responses to estimate overall application availability and performance (Active Monitoring).
- **Client agent:** Apply client based agent technology to monitor client Web browser and system activity to derive end-to-end response time or utilize application instrumentation to measure response time.

Knowing that a response time problem exists is an important first step in implementing a corrective action. For Web applications some examples of possible corrective actions are as follows:

- An installation may decide to relocate parts of the Web server equipment (Proxy servers) nearer to problem areas to reduce network latency [3].
- Capacity and/or technology upgrades along the transaction path may be driven based upon the service delivered.
- The installation may decide to develop adaptive applications [4] capable of using the recorded response time information to vary resource demands and consequently mitigate the impact of network or server latency.

But for any or all of these and other corrective actions to occur, it must begin with measurement. WEBARM represents an easy to implement, minimal overhead, and reduced maintenance cost approach to Web application response time measurement. It provides a set of instrumentation APIs, similar in nature to the ARM Version 1.0 APIs, but without the complexity. Deployment, often a barrier to agent implementation, is also addressed through WEBARM's mobile code based agent architecture.

This paper is organized as follows. Section 2 provides an overview of various Web response time monitoring techniques commonly used to capture or estimate response time, and introduces our focus technique, the Client Based Mobile code agent technique (WEBARM) which is evaluated in the remainder of this paper. In section 3 we provide an in depth description of the WEBARM technique used to capture and record Web application response time. Section 4 presents the results of measurement experiments designed to assess the overhead involved with the response time measurement technique presented in section 3. And finally, a summary and conclusion are offered in section 5.

2 Web Application Response Time Measurement

In this section, we examine commonly used approaches to Web application response time measurement. Considering an application's life cycle stage, measuring response time under various loads during development is an essential part of application per-

formance engineering. However, equally if not more important and sometimes neglected is measurement of applications during the production stage. It is at this point where application users including customers are directly affected by application performance. It is at this point where knowing and adapting to changing infrastructure conditions begins with knowing the actual service delivered. We have identified three Web Application Response Measurement techniques: Server based measurement, Site Monitoring, and Client Agent based measurement. Each of the presented techniques measures overall response time and the components thereof in varying degrees of completeness and accuracy.

2.1 Server Based Measurement Techniques

A common approach to Web application response time measurement is to use the Web Server logs to measure Web page references; bytes transferred, and server time [5]. However, the measurements are server focused and consequently do not capture the entire end-to-end response time. Server logs typically include per page or object measurements, so where multiple browser objects are requested server response time estimations must consider the first to last object timings. Further, only objects not cached at the client browser are included in the server logs complicating the estimation of response time.

If the server application is instrumented, server focused measurements can be enhanced with a *ping* or *traceroute* measurement to estimate the network latency. By applying a fixed client time the overall end-to-end transaction response time is estimated. Alternatively, where server measurement is not possible, network "Sniffers" or monitors may be used to measure Web server response time calculating the difference between connect and disconnect times or page service time. Web server measurements captured by operating system facilities and/or by network monitoring collect a wealth of information regarding the performance, availability, and reliability of the Web server and hosted applications. Useful measurements, but Server Based Measurements do not provide a true assessment of end-to-end response time.

2.2 Site Monitoring Techniques

As commercial enterprises become more dependent upon the Internet, external site monitoring services have emerged as an approach to Web application response time and availability monitoring. This involves the strategic placement of a set of application monitoring probes designed to periodically execute a scripted reference to a Web site's home page or, in a more involved procedure, invoke a series of Web site transactions[6][7]. The script timings are recorded for response time analysis, availability trending, and problem alerting. Some implementations involve placement of probes connected to specific locations throughout the Internet to enable assessment of regional variations. This approach facilitates the understanding of how accessible /available the specific Web site and how responsive the site is likely to be. This technique is also useful for Intranet sites within and organization's firewall. However, it does not capture the true end customer experience. End user PC processor capacity, memory, hardware/software platform, and modem access speeds vary widely and may

play a significant role in the end user performance. Regarding response time, what is measured from a probes perspective is the probes view of the Web site or application and this may or may not be representative of actual end-user equipment and experience. While clearly a useful and valuable approach to performance and availability measurement, site monitoring is not a complete solution since it does not capture the end-customer's experience.

2.3 Client Agent Monitoring Techniques

This technique involves the application of client agent technology to locally monitor Web browser activity and calculate response time. Two forms are prevalent, installed agents [8] and mobile code/agents [9]. Installed agents exist as services on the end-user's equipment. The client agent monitors Web browser activity for transaction start/stop events and then logs the transaction response time information either locally at the client or to an external server. The difficulty is that an agent must be installed and active to monitor the Web application. Since the Web client machine is often beyond the control of the organization, the ability and justification involved in deploying agents on the customer's machine represents a significant implementation challenge. Moreover, the Client agent must be parameterized to intercept the transaction start and stop events; application changes may disturb the agent's understanding of transaction events and thus, it is likely the agent must be retrained or parameterization for each application change.

Mobile client agent code moves with the Web application to the target machine [9]. The mobile agent code is directly integrated into the application's Web page. Since the agent code is integrated with the application, interception of transaction start and stop events is accomplished through instrumentation. In addition, since a "light-weight" end-customer presence is desired, recorded response time information is logged directly to an external server. The advantages are that no formal agent deployment is needed and consequently remote maintenance costs are reduced. The disadvantage is that the mobile code must be transmitted at least once to be cached locally at the end-customer's workstation, and depending how frequently the local browser cache is purged impacts how frequently this overhead is incurred.

3 WEBARM – Mobile Code Based Web Response Time Measurement

In this section, we provide an in-depth examination of the client based mobile code agent (WEBARM) technique introduced in section 2. As indicated, the client based agent alternatives are to deploy an agent service/daemon to capture transaction start and stop events or integrate the response time measurement instrumentation into the application's Web pages to capture transaction response time.

3.1 WEBARM Relationship to the ARM API Standard

For instrumentation we use a set of Application Programming Interface (API) functions that are a simplified adaptation of the ARM Application Response Measurement

API [10]. We define three API functions *ARMstart*, *ARMstop*, and *ARMwrite*. The *ARMstart* API function records the start or beginning of a Web transaction by storing the start time in a frame based array variable or a cookie. The *ARMstart* API, as we present it here, combines the ARM version 1.0 *arm_init*, *arm_getid*, and *arm_start* API calls into a single function call [11]. *ARMstart* is typically invoked in reaction to a transaction event such as a "mouse click" or "function key". The *ARMstop* API function records the transaction stop or end-time and calculates the response time. *ARMstop* emulates the *arm_stop* ARM API to the extent that it records the stop time and computes the response time; however, the response time logging process is built as a separate function to allow the Web application the freedom to select when to log the response time information. *ARMstop* is invoked at a point in a Web page representing a transaction end point. The last WEBARM API defined is *ARMwrite*. While there is no direct ARM equivalent, the function is normally embedded in the *arm_stop* API. *ARMwrite* directs the recorded transaction information to a logging server and should be coded at a point in the Web page processing that overlaps the log processing with user think time.

3.2 Timer Storage Alternatives

To facilitate response time determination, an important consideration is the storage of timer information between the *ARMstart* and *ARMstop* API calls. Two alternative techniques are presented: the Cookie method and the Array-Frame method.

- **The Cookie Method:** Client Web browser "cookies" are typically used to maintain state information locally at the client. For example, a Web application may collect some information in a session to be used in subsequent sessions. To avoid re-requesting the information from the end user, the entered data is stored in temporary storage, referred to as a "cookie", and later retrieved as needed. Rather than store end-user information, the WEBARM Cookie method uses cookies to store the transaction start time as part of the *ARMstart* API (Figure 1). *ARMstart* is normally called in response to a "submit button" or other event designating the start of a transaction. The *ARMstop* API retrieves the transaction start time from a specific cookie, calculates and returns the response time. The response time is then logged to a server using the *ARMwrite* API. The advantages of this method exist predominately in that a cookie can store information across browser sessions and instances. The main disadvantage is the slightly higher overhead involved in comparison to the Array-Frame technique.
- **The Array-Frame Method:** As opposed to using cookies to store the transaction start time, the Array-Frame method utilizes a "top-frame" JavaScript array variable to store the transaction start time and identification information. Similar to the cookie technique, the *ARMstart* API saves the transaction start time, but rather than in a "cookie" an array variable is used (Figure 2). *ARMstop* retrieves the start time from the array, calculates, and returns the response time. The *ARMwrite* API is then invoked to log the ARM data to a server. The advantage is the lower overhead and simplicity of implementation. The main disadvantage is that access to the stored time is not persisted across sessions and a separate frame is needed to store the start time value.

3.3 The WEBARM Instrumented Web Page

The WEBARM APIs and supporting functions are included and referenced in the Web page by the programmer during development, as a set of embedded functions or as a separate object on the Web page. However, using a separate object allows the WEBARM functions to be cached locally at the client, and subsequent references can avoid the network transfer time for each reference from the cache.

The HTML statements below illustrate the separate object approach to instrumentation, in a basic two-page Web application. Page 1 (Figure 3) is an example Web page including a basic HTML form and the associated "submit" button to initiate the form processing. When activated, "submit" generates the `onClick` event and invokes the `ARMstart` function to store the current time in a timer identified as `ARM_Timer`. Next, our sample application opens `Page2.html` (Figure 4) in the same window to complete the transaction.

The JavaScript object `ARMn.js` contains the WEBARM APIs, supporting functions, and the in-line code to invoke the `ARMstop/ARMwrite` APIs to compute, log, and display the response time. Note each page must reference the same `ARMn.js` object to facilitate browser caching, and "n" represents the method employed, for example, `ARMCookie.js` or `ARMArray.js`.

Fig. 3. Sample Web Page 1

```

Filename: Page1.html

<html><head>
<title>Test Web App</title>

<script lan-
guage="Javascript1.2"
SRC="/ARMn.js"> </script>

</head><body>

<FORM>
<INPUT TYPE=button
VALUE="Submit"
on-
Click="ARMstart('ARM_Timer');
window.name='test';
win-
dow.open('Page2.html','test') ">
</FORM>

<p> The submit button records
the start time and opens page2
in the same window.
</body>
</html>

```

Fig. 4. Sample Web Page 2

```

Filename: Page2.html

<html><head>
<title>Test Web App</title>

<script lan-
guage="Javascript1.2"
SRC="/ARMn.js"> </script>

</head><body>

<p> Web page returned, in-
cluding response time.
</body></html>

```

Fig. 5. Frames Example

```

Filename: Page0.html

<HTML><HEAD>
<TITLE>ARM Test Frame
</TITLE></HEAD>

<FRAMESET ROWS="1,*">
  <FRAME SRC="dummy.html"
NAME="FR1">
  <FRAMESET >
    <FRAME SRC="Page1.html"
NAME="FR2">
  </FRAMESET>
</FRAMESET>
</HTML>

```

The Array-Frame technique requires the use of frames to persist the recorded timers across Web page references. Without the "top frame" storage area, the timer information is cleared when a subsequent HTML document is loaded. Figure 5 provides an example page using frames to reference the sample Web application. Note the frames page (Page0.html) must be loaded first which in turn references and loads Page1.html.

3.4 WEBARM Transaction Logging Alternatives

Once the response time data has been collected and communicated to the Web server, the information must be logged to disk to facilitate analysis and reporting. The logging alternatives considered for WEBARM are as follows:

1. **Web server logs:** Web servers typically log information for each server access, including HTML page references, image loads, and script executions. In the case of (IIS) Internet Information Server (Microsoft's Web Server), the invocation of

the CGI logging script (ARMweblog), called during the *ARMwrite* API, is logged including all the parameters used to invoke the CGI – notably response time.

2. **CGI Script logging:** Since the *ARMwrite* API invokes a script (ARMweblog) to communicate the response time information to the Web server, the passed CGI parameters are then logged to disk under the control of the ARM Web logging script. We evaluate the processing impact associated with alternatives 1 and 2 in section 4.3.

3.4.1 WEBARM Log Collection

For many mission critical Web application implementations, parallel scalable Web clusters[12] are configured to enhance availability, performance, and scalability. However, this creates a design challenge for WEBARM, the separate Web server logs must be merged to develop a complete assessment of the application's performance.

Consequently, we developed a separate asynchronous process, known as the WEBARM Collector (Figure 6), to periodically collect the logged data from the clustered servers, into a consolidated database for analysis and reporting purposes. Moreover, the WEBARM Collector enables the periodic issuance of performance threshold alerts based upon the collected response time information.

Selecting a flexible log format and database architecture is an important related design consideration; [13] [14] describe some recent log format research. WEBARM utilizes a keyword based log communication format and a variable length delimited log record when stored in the WEBARM database.

3.5 WEBARM JavaScript Implementation

The mobile agent/code implementation technique described herein utilizes a set of JavaScript functions to implement the ARM APIs and related supporting subroutines packaged in a browser cacheable object. For the Cookie method we have adapted and renamed some commonly available cookie management functions [15]: *setCookie* and *getCookie*, to support timer storage and access (*setTimer* and *getTimer*). All the functions are contained, in the Web page included ARMCookie.js/ARMArray.js object (see section 3.1) depending upon which alternative timer storage method is employed.

The "*setTimer*" function, as its name implies, is used to set a named cookie or array element to a specific value. For WEBARM, we store the Web application's transaction start time. "*setTimer*" requires two input parameters: timer-name, and the as-

sociated timer value. The other support function -"*getTimer*" is used to access the stored timer value i.e., the transaction start time after the transaction completes processing or when needed by the *ARMstop* API. "*getTimer*" requires that the timer-name be passed as an input argument.

The individual API functions, support subroutines, and in-line code included are presented below:

1. **ARMstart:** This API (Figure 7.) stores the current time in a specifically named timer using the *setTimer* function. By default the name ARM_Timer is used; however, it is possible to use any unique name.
2. **ARMstop:** This API computes the transaction response time by, obtaining the current time, accessing the stored cookie or array start time value, and then computing the response time. Note the *setTimer*(timerName, 0) effectively deletes the timer. It was discovered, when using the Cookie method, that by removing a cookie through the cookie delete function increased the overhead involved. Cookie functions that employ the expiration date processing "expires=" experience significantly higher overhead than functions avoiding this parameter (see section 4.1.1).

<p>Fig. 7. ARMstart</p> <pre>function ARMstart(timerName) { setTimer (timerName, new Date ().getTime());}</pre>	<p>Fig. 9. ARMwrite</p> <pre>function ARMwrite(value){ log_url="http://" + document.domain + "/scripts/armweblog.exe?"; cgi= new Image(); cgi.src=log_url+" "+escape(value); return true;}</pre>
<p>Fig. 8. ARMstop</p> <pre>function ARMstop(timerName) { endtime = new Date().getTime(); starttime=getTimer(timerName) If (starttime==0 endtime == starttime){ resp = 0} else{ resp = (endtime - starttime)/1000} setTimer (timerName, 0); return resp;}</pre>	

3. **ARMwrite:** This API logs the measure transaction response time to the initiating Web server using a technique that passes the response time information along with an Image object CGI request. This technique eliminates the requirement to have a signed script or a Java Applet to return log data to the Web server. Since the response time information is normally logged in the Web server log, the CGI program can be designed to simply return a null image pointer, or for more sophisticated implementations special processing can be performed within the CGI to store additional measurements. For example, the IP address of the sending workstation or client could be translated to the corresponding machine name or a network *traceroute* command could be executed to record network round trip times to assess network conditions near the time of the transaction execution.

Fig. 10. getTimer Cookie Method

```
function getTimer (name) {
  var result=0;
  var myCookie= " " + document.cookie + ";";
  var searchName = " " + name + "=";
  var startOf-
Cookie=myCookie.indexOf(searchName);
  var endOfCookie;
  if (startOfCookie != -1){
    startOfCookie += searchName.length; endOf-
Cookie=myCookie.indexOf(";",startOfCookie);
    result=unescape(myCookie.substring(
startOfCookie, endOfCookie)); }
  return result;}
```

Fig. 11. getTimer Array Method

```
function getTimer (name) {
  return(getTimers()[timerName]);}

function getTimers() {
  var result = top.ARMTimers ;
  if (result == null) {
    top.ARMTimers = new Array() ;
    result = top.ARMTimers ;    }
  return result ;}
```

4. **getTimer:** The *getTimer* function is used to access and return the value associated with a specifically named timer. The *ARMstop* API calls *getTimer* to access the necessary start time to compute the transaction response time. The cookie and array-frame methods are illustrated in figure(s) 10 and 11 respectively.
5. **setTimer:** *setTimer* stores a value in a specifically named cookie. Three parameters are used: the cookie name, the value to set, and an expiration date. As indicated above, we avoid the use of expiration date to delete a cookie to reduce overhead

Fig. 12. setTimer Cookie Method

```
function setTimer (name, value, expires) {
  if (!expires) {document.cookie = name + "=" +
escape (value) + "; path=/";}
  else{document.cookie = name + "=" + escape
(value) + ";
  expires=" + expires.toGMTString() + ";
path=/";} }
```

Fig. 13. setTimer Array Method

```
function setTimer (name, value, expires) {
  getTimers()[timerName] = value;}
```

Fig. 14. In-line Code

```
resp=ARMstop("ARM_Timer");
if (resp>0 ){
  document.writeln("Response Time:" +
resp);
  ARMwrite("resp=" + resp + ";" +
"tran=" + document.title + ";" +
"etime=" + (new Date()).getTime() + ";");}
```

6. **In-line code:** Optionally this program code can be placed with in the *ARMn.js* object (as shown) or maintained separately in the Web page. This code is placed within the Web page to reflect the end of the Web transaction, normally at the top of the document. First, *ARMstop* is called to compute the last transactions response time. Second, if the response time returned is valid the *ARMwrite* function is called to log the information at the Web server. Note the format of the logged information consists of a string of **KEYWORD[x]=value;** pairs - similar to the Universal Logging Message format described in [13]. The [x] optionally defines

a unique index number to enable multiple transactions to be logged in a single communication.

4 WEBARM Instrumentation Performance Analysis

To assess the impact of the WEBARM instrumentation, a series of benchmark experiments were conducted on the following platform configurations to evaluate the processing delays an application may experience due to instrumentation.

1. Intel Pentium II 266MHz, NT 4.0 SP6, Internet Explorer 4.0
2. Intel Pentium II 266MHz, NT 4.0 SP6, Internet Explorer 4.0, via dial-up 33.6bps modem.
3. Intel Pentium II 350MHz, NT 4.0 SP6, Internet Explorer 4.0
4. Intel Pentium II 450MHz, NT 4.0 SP6, Internet Explorer 4.0
5. Intel Pentium II 450MHz, NT 4.0 SP6, Internet Explorer 5.0

With the exception of configuration 2 above, a 100Mbps Ethernet LAN connection to the Web server was used. The Web server configuration consisted of a dual 466MHz Celeron processor configuration running NT 4.0 and IIS 4.0 at SP6. Configurations 3-5 were connected to a dual 450MHz Intel Xeon processor configuration running NT 4.0 IIS 4.0 SP4 via a 100Mbps Ethernet LAN. Performance measurements and analysis are presented for the "Cookie" and "Array-Frame" techniques (described in section 3) in 4.1 and 4.2 respectively. And finally, we present a volume stress test benchmark of the "ARMweblog" program in 4.3 to assess logging impact and efficiency.

4.1 Performance Analysis of the Cookie Technique

To measure overhead of the Cookie technique a simple form based, test Web application was developed and instrumented. Measurements include the overall impact on a set of instrumentation API calls to assess impact per transaction, individual function call timings, and the recorded test application response times.

4.1.1 Cookie Technique Individual Transaction Overhead Assessment

The objective of this experiment is to measure a complete set of instrumentation calls for a single transaction including the JavaScript interpretation time. Assuming the script (ARMn.js) is loaded from the browser cache, this benchmark assesses the expected impact each instrumented page should experience. Figure 15 below illustrates the various measurements observed across the configurations evaluated.

The 450Mhz Pentium II processor NT 4.0 IE5 configuration recorded the lowest instrumentation impact at 1.20ms and the 266 MHz Pentium II the highest measured impact. One result in particular is interesting. The initial *ARMstop* API included a function to delete the stored cookie by setting the expiration date to a specific past date. This is a frequently used approach to cookie cleanup; however, initially a significant 61% greater path-length overhead was observed for the 450MHz IE4 configuration in comparison to the 450MHz Pentium II running IE5. By removing the "ex-

pires=" cookie parameter and changing the *ARMstop* function to recognize a zero as a deleted cookie, path-length overhead was reduced to - a 7.6% differential between configurations 4 and 5. This suggests that performance improvements have been implemented in IE5 from the IE4 browser.

4.1.2 Cookie Technique Individual API Measurements

This benchmark scenario involves execution of each API (*ARMstart*, *ARMstop*) function 1000 times for 10 iterations (figure 16). As indicated, the functions measured include the *ARMstart* function, which stores the transaction start time in a cookie, and the *ARMstop* function, which reads the cookie and calculates the response time. The *ARMwrite* sub-function measures the path-length cost of logging the recorded response time to the Web server. This function was also invoked 1000 times and the mean path-length computed.

The Pentium II 450MHz IE4 configuration exhibits the best *ARMstart* and *ARMstop* performance while the PII 450MHz IE5 recorded the most favorable *ARMwrite* timing. The 40% reduction in path-length time for the *ARMwrite* function from the PII 450MHz IE4 to the IE5 configuration is also significant. Note that the reduction in path-length between IE4 and IE5 occurs predominantly in the *ARMwrite* function.

4.1.3 Cookie Technique Test Application Response Time

The test Web application used to evaluate the performance impact of the Cookie technique consists of two Web pages. The first page, page 1, contains a form with a button control to initiate the response time measurement (*ARMstart* function) and then open a second page in the same window. The second page, page 2, calls the *ARMstop* function and displays the response time and function path-length timings, demonstrating the ability to persist timer information across pages. Figure 17 examines the relative performance of all the configurations executing the test Web application.

Results are generally consistent with the single transaction impact, except that the 266MHz modem configuration is showing at least a 53% increase in response over the 266MHz LAN configuration.

An important design consideration is that the ARM program logic was configured as a separate script object (ARMn.js) in the test Web application. The overhead to access the script initially (non-cached) is .25 seconds for the LAN based configurations and .7 seconds for the dial-up 33.6bps modem. The alternative to the "separate script entity" is to integrate the (ARMn.js) script into the application's Web page. However, while the initial Web-page response time is reduced, subsequent arm script references on different pages do not benefit from (ARMn.js) script caching. In addition, if the script is integrated into the application page, script changes will require all pages with the integrated script to be updated; whereas, if the separate entity (ARMn.js) is changed no application page updates are required. Next, we review the performance of the "Array-Frame" technique.

4.2 Performance Analysis of the Array-Frame Technique

To measure overhead of the "Array-Frame technique" a "frames" based, test Web application was developed and instrumented. As with the Cookie technique analysis, measurements include the overall impact on a set of instrumentation API calls to assess impact per transaction, individual function call timings, and the recorded test application response times.

4.2.1 Array-Frame Technique Individual Transaction Overhead Assessment

As with 4.1.1, this measurement describes the expected impact (using the Array-Frame technique) each instrumented page should experience given that the script is loaded from the local cache. Figure 18 illustrates the various measurements observed across the subject configurations. The 450MHz Pentium II IE5 configuration recorded

the lowest instrumentation impact at (.7ms) and the 266MHz Pentium II the highest measured impact (4.3 ms). The main difference between the Cookie and Array-Frame techniques is the location where the measurement timings are stored, i.e., cookie versus memory array.

4.2.2 Array-Frame Technique Individual API Measurements

Similar to the experiment described in 4.1.2, this benchmark scenario (Figure 19) involves execution of each API (*ARMstart*, *ARMstop*) function 1000 times for 10 iterations. As indicated, the functions measured include the *ARMstart* function, which stores the transaction start time in an array variable, and the *ARMstop* function, which access the array variable and calculates the response time. The *ARMwrite* sub-function measures the path-length cost of logging the recorded response time to the Web server. Although the Array-Frame technique supports multiple concurrent timer measurements, a single response was used for our sample application.

4.2.3 Array-Frame Technique Test Application Response Time

The Array-Frame test application employs a Web page consisting of three frames:

1. Frame 1: The Top frame is used to store the ARM measurement array.
2. Frame 2: A Web page is loaded in frame 1 that contains a form, similar to the "cookie" test Web application, with a submit button. The "submit" button invokes the *ARMstart* function and opens an instrumented page in frame 3.
3. Frame 3: When the Web page is loaded a call is made to the *ARMstop* and *ARMwrite* functions to display and record the measured response time, function path-length timings, and demonstrate the ability to persist timer information across frames.

Figure 20 examines the relative performance of all the configurations executing the test Web application. As with the Cookie technique, the 450MHz Pentium II configuration exhibited the superior performance with a .02 seconds response time.

Comparing the Array-Frame to the Cookie technique, in general, response times are equal or better (for the Array-Frame technique), ranging between .0 and .12 seconds difference. The Cookie technique path-length timings are consistently less than the Array-Frame for configurations 1 to 4, and the Array-Frame shows the improvements for the Pentium II 450MHz configuration.

4.3 Web Logging Capacity Requirements

To assess WEBARM logging capacity requirements, we developed a benchmark program to repeatedly execute the "ARMweblog" CGI script at various rates while recording the corresponding Web server processor utilization. The results of the "ARMweblog" benchmark for the dual 466MHz Celeron processor configuration indicate that in the range of 1 to 25 logs per second, processor usage increases linearly from 1.9% to 52%. The alternative, IIS logging technique, extracts the response time information directly from the IIS (Web Server) logs. The results obtained indicate that by increasing the transaction rate from 1 to 25 logs per second processor impact varied from .1% to 3.5%. Note the significant difference in overhead required to spawn and execute the CGI process in comparison to basic Web server logging. Web server logging is clearly the low overhead alternative if additional processing or metric collection is not required.

5 Summary

Web application performance is an important concern for businesses implementing mission critical applications over the Internet, since the level of performance delivered directly affects the customer. In order to understand the level of service an application is providing measurement is essential. We presented three forms of Web application performance measurement methodologies: server based response time measurement, site monitoring, and client agent/based measurement.

We provided a detailed design and software description of WEBARM a mobile code based agent application instrumentation tool used for capturing end-to-end response times. Although the application designer must be concerned with the additional API calls required, the end result is a more manageable application through service measurement.

We have analyzed several important design aspects of the WEBARM agent software. In particular, WEBARM agent architecture, Web page instrumentation, logging considerations, and log collection techniques were studied. We proposed two methods of storing timer information across Web page references: the Cookie and Array-Frame methods, and provided a detailed description of the WEBARM APIs and supporting functions. We proposed, developed, and benchmarked the WEBARM agent techniques and studied the impact of the WEBARM logging CGI process.

WEBARM is an adaptation of the ARM Version 1.0 API set; however, WEBARM has been optimized and simplified for Web applications. While ARM Version 3.0 incorporates JAVA support, its emphasis is more on JAVA applications than Web page instrumentation [16].

Our future work includes continued monitoring of the ARM APIs evolution, investigation into response-time component analysis for Web applications, and network based designs for automating server instrumentation.

References

- [1] Nelson, M., "Fast Is No Longer Fast Enough", *Information Week*, June 2000, <http://www.informationweek.com/789/web.htm>
- [2] "Web Commerce Doubles in Year 2000 As Global Growth Continues," ActiveMedia Research, August 2000, http://www.activmediaresearch.com/body_free_newsroom.html
- [3] Caceres, R., et al., "Web Proxy Caching: The Devil is in the Details", *ACM SIGMETRICS Performance Evaluation Review*, Vol. 26 No. 3, (December 1998): pp 11-15.
- [4] Gross, T., Steenkiste, P., Subhlok, J., "Adaptive Distributed Application on Heterogeneous Networks", *Proceedings of the Eight Heterogeneous Computing Workshop*, (1998).
- [5] Hochheiser, H., Shneiderman B., "Understanding Patterns of User Visits to Web Sites: Interactive Starfield Visualization of WWW Log Data", *Institute for Systems Research Technical Report*, (March 1999).
- [6] Barford, P.,Crovella, M., "Measuring Web Performance in the Wide Area", *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27 No. 2, (September 1999): pp 37-47.
- [7] Norris, C., "Understanding Web Site Speed and Reliability", *Keynote Systems, Inc.*, June 2000, http://www.keynote.com/services/downloads/product_lib.html
- [8] Kitay, D., "Understanding End-To-End Monitoring", *Candle Computer Report*, Nov-Dec 1997, Vol.19 No. 9, Candle Corporation
- [9] Pham, V., Karmouch, A., "Mobile Software Agents: An Overview", *IEEE Communications*, July 1998, pp. 26-37.
- [10] "ARM 2.0 SDK User's Guide", Computer Measurement Group, November 1997, <http://www.cmg.org/regions/cmgarmlw/sdk/armguide.html>
- [11] Elarde, J., Brewster, G., "Performance Analysis of Application Response Measurement (ARM) Version 2.0 Measurement Agent Software Implementations", *Proceedings from the 19th IEEE International Performance, Computing, and Communications Conference*, 2000
- [12] Microsoft TechNet, "System Redundancy", April 2000, <http://www.microsoft.com/technet/ecommerce/sysredund.asp>
- [13] Tierney, B., Johnston, W., et al., "The NetLogger Methodology for High Performance Distributed Systems Performance Analysis", *Proceedings from the Seventh International Symposium on High Performance Distributed Computing*, 1998.
- [14] Park, D., "XLF: The Extensible Log Format", XLF Working Group, July 1998, <http://www.docuverse.com/xlf/NOTE/xlf-19980721-all.html>
- [15] JavaScript Unleashed, Richard Wagner, et al., Sams.net Publishing, 1997, Second Edition
- [16] Johnson, M., " ARM 3.0 – Enabling Wider Use of ARM in the Enterprise", December 1999, <http://www.cmg.org/regions/cmgarmlw/ARM30.html>