Delivering Adaptive Web Content Based on Client Computing Resources

Andrew Choi and Hanan Lutfiyya

Department of Computer Science
The University of Western Ontario
(choi@csd.uwo.ca, hanan@csd.uwo.ca)

Abstract. This paper describes an approach to adapting Web content based on both static (e.g., connection speed) and dynamic information (e.g., CPU load) about a user's computing resources. This information can be transmitted to a Web Server in two different ways. XML is used so that there is one copy of the content, but multiple presentations are possible. The paper describes an architecture, prototype and initial results.

1 Introduction

The users of electronic commerce applications have certain expectations about the Quality of Service (QoS) provided. By QoS, we are referring to non-functional requirements such as performance or availability. An important measurement of Quality of Service is "key to glass" response time. This refers to the time that passes after a user clicks the mouse button or presses the return key to submit a request to a WWW server to the time that the results of that request are displayed on the monitor glass. Electronic commerce retailers (e-tailers), such as Amazon.com, Chapters.ca, and other dot-coms, recognize that the reference to the WWW as being the "World Wide Wait" is a major impediment in the growth of electronic commerce applications.

To date most of the work for improving Quality of Service (QoS), has revolved around the web server and the network. There has been little work on the client side despite the research [4] showing the importance of the client computing resources especially the network link from the client machine to the Internet. This paper focuses on the client side. The approach taken is based on findings that show that web pages that were retrieved faster were judged to be significantly more interesting then web pages retrieved at a slower rate. Other findings in [2] indicates that the user prefers web pages to be progressively displayed rather than web pages that are displayed at a slower rate, but may have more graphics at a higher resolution. This provides the user with feedback concerning the web server, knowing that the downloading process is still taking place. The conclusion is that that the response time affects a user's impression of a web site.

Let us now look at the following scenario: Let us assume that we have two users: user *A* is using a slow computer with a phone modem for connection to the Internet and user *B* is on a fast computer with a cable modem for connection to the Internet.

M. Reed Little and L. Nigay (Eds.): EHCI 2001, LNCS 2254, pp. 337–355, 2001.

The difference between the response time of these two users, assuming the network and web server loads are identical, relies on client computing resources. To state the obvious, user *B* receives the content faster than user *A*. This difference in client computing resources has forced content providers [1] to provide a compromise version of content that hopefully will not place an undue burden on clients with fewer computing resources, yet will remain satisfactory for clients with the high-end in computing resources. One approach is to have different content delivered to different clients [1]. Currently, this requires that for a single content provider would need to maintain multiple versions of the same web site. Our research addresses this problem. We provide a solution that allows a web site to have a single copy of the content, but provides a different presentation of that content based on the client's computing resources. The main advantage of this approach is the elimination of unnecessary multimedia content being delivered to clients who do not have the necessary computing resources that best supports this content.

The following sections of this paper will present the research that we have done to date in improving a client's "key to glass" response time. Section 2 provides the design architecture, followed by the implementation in Section 3. A set of experiments in Section 4 shows the improved "key to glass" response time for clients. In Section 5 we will examine a few other techniques in improving response time. Finally, we shall provide some concluding thoughts in section 6.

2 Architecture

The architecture (graphically depicted in Figures 1 and 2) entails two primary components: the client and server components. The client component (which includes the Web Browser) is used to collect client computing resource information. Computing resource information includes, but is not limited to, the type of processor, type of Internet connection, percentage of processor load and percentage of memory being used. There are two approaches to getting this information to the web server. The first is a *push* approach where the information accompanies each web request to a web server. The second is a *pull* approach where the information is requested by the web server when needed. Upon receipt of the client request, the server component (which includes the Web Server) is responsible for analyzing the client computing resource information to determine an appropriate presentation descriptor that is to be applied to the content being requested by the client. We will now describe these components in more detail.

2.1 Client Component Architecture

The Client Component of the architecture is graphically depicted in Figure 1.

The Client Component collects the client computing resource information that is needed by a Web Server for processing a client request. We refer to this as *QoS Data* and it may include the following: (i) *CPU Type*: This represents the CPU speed of the client device that is requesting the Web page. (ii) *Connection Speed*: This measures the connection speed of the client device to the Internet. (iii) *CPU Load*: This represents the amount of processor utilization on the client device. (iv) *Memory Load*:

This represents the amount of total memory, both physical and virtual memory, being used on the client device (percentage).

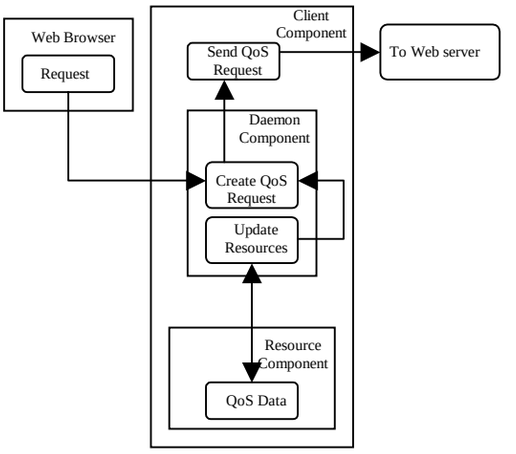

Fig. 1. Client Components

The two components involved in collecting *QoS Data* and including it in a web request are the Daemon and Resource components. In the push approach the Daemon converts a web request from any web browser into a request that includes *QoS Data*. In the pull approach, the Daemon process maintains (i.e., stores) the *QoS Data*. It provides an interface to other processes to retrieve this data. The *QoS Data* dynamically changes over time. The Resource Component is responsible for monitoring the computing resources to provide the information needed in *QoS Data*. It provides an interface to allow for monitoring of the necessary resources. It must be noted that this is just an interface. The actual implementation is based on the specific client device.

2.2 Server Component Architecture

The Server Component of the architecture is graphically depicted in Figure 2. The RequestHandler component is responsible for the initial processing of a request that is received by the web server. It extracts the *QoS Data* (done differently for the push and pull models). It passes the *QoS Data* to the QoS component, which encapsulates an algorithm that uses this data to determine an appropriate *classification* value. More specifically it encapsulates the order of operations (some are calls to the interface of the Classification component). The Classification component maintains the following: information needed to determine the classification value (described in the next paragraph) and a mapping from a classification value to a *presentation descriptor* value. The *presentation descriptor* value is applied to the web content that is to be returned to the client.

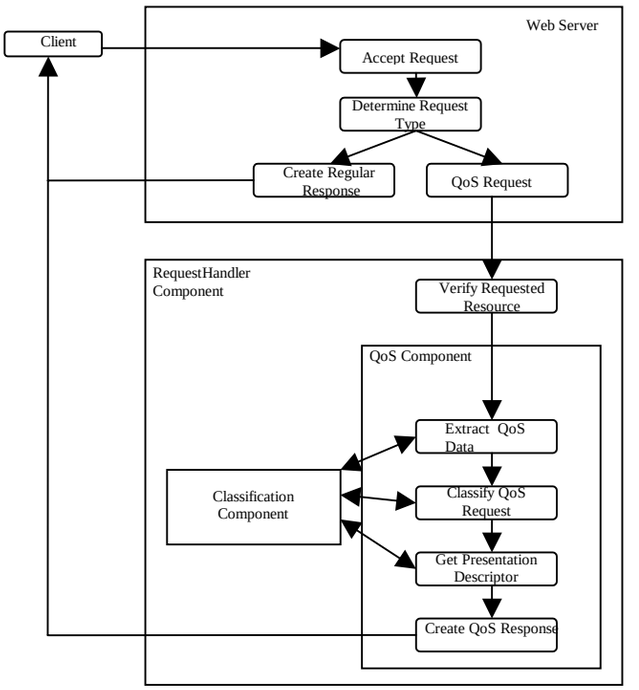

Fig. 2. Server Components

The algorithm used for computing the *classification* value is briefly described as follows. Each of the data items in *QoS Data* (*CPU Type*, *Connection Speed*, *CPU Load* and *Memory Load*) are assigned weights to reflect their contribution to the "glass response time". For example, the weight assigned to *CPU Type* could be 30%, for *Connection Speed* it could be 30%, for *CPU Load* it could be 30% and for *Memory Load* it could be 10%. This weighting assumes that the type of the CPU, the connection speed and the load on the client machine contribute equally and much more than memory load. A weight is applied to a *category* value. We will use *CPU Load* to illustrate what we mean by category value. One possible categorization is based on having two categories: CPU loads that represent less than 50% utilization and CPU loads that represent 50% or more utilization. We know that if there is higher utilization then there will be fewer CPU cycles on the client device to process the response. Thus, resulting in the end result of a slower response time. We can assign a *category* value to each of these categories. The appropriate *category* value is determined and is multiplied by the weight associated with *CPU Load*. The sum of these values is the *classification* value The *classification* value is mapped to a *presentation descriptor* value which is then applied to the content. The result of this application is what gets sent back to the client.

2.3 Interactions

This section is intended to examine the flow of a request from the client device to the final delivery of a response from the web server. We assume that the specified resource in the request is available**.**

1. The request is made in the web browser.
2. In the *push* approach, the request is intercepted from the web browser by the Daemon component. The Daemon component adds the *QoS Data* to the request. In the *pull* approach, the request is sent through without any additional modifications (in the current prototype, a web request is considered QoS enabled if it uses the ".xml" extension). The *QoS Data* used by the Daemon Component is updated periodically (the period is relative to the speed of the processor) by the Resource component.
3. When a request arrives at a web server, the Request component determines the nature of the incoming request i.e., whether the request is QoS enabled or not. In the *push* model, the Request component checks to see if the QoS Data is appended to it. In the *pull* model, it checks to see if the request uses the ".xml" extension. If so, the RequestHandler component will retrieve that data from the client. If the request does not include *QoS Data* or does not use the ".xml" extension, the web server will immediately process the request; otherwise the request is forwarded to the QoS component.
4. The QoS component processes the *QoS Data* to determine the appropriate classification value based on the brief description of the algorithm presented in the previous section. It encapsulates an algorithm that has operations that make calls to the Classification component.
5. The QoS Component takes the *classification* value and requests from the Classification component the associated *presentation descriptor* value.
6. The URL for the web content requested by the client is processed with the returned *presentation descriptor* value. Based on this processing, the appropriate QoS enabled response is produced for the client device. This is very critical in our process. If the client device was a wireless cell phone, it would not be capable of viewing images or handling HTML content. Hence an appropriate *presentation descriptor* value would be required to convert the web content into Wireless Markup Language (WML) content, so that the client device could display the results.
7. The response with the web content produced from the previous step is returned to the client device that made the request. The client device's web browser then processes the web content and displays it accordingly.

If there was any problem with the *QoS Data*, then default classification values are used (in our prototype we defaulted to classifications resulting in the simplest content presentations).

3 Implementation

In this section we describe the prototype implementation. Open-source code (Mozilla) is available for the Netscape browser, but the source code for this browser consists of over four million lines of code. Due to this complexity, it was decided to first implement the push approach. This was done using the Remote Method Invocation (RMI) API provided by Java.

3.1 Client Component

This section describes implementation details associated with the Client Component.

3.1.1 Daemon Component

We implemented the pull approach for the retrieval of client computing resource information. The Daemon component is implemented using a Java class that provides a listen queue for monitoring incoming requests for client computing resource information from the web server. The Java RMI enables the web server to make a method call to the remote object as if the remote object were local, without the need to explicitly program the socket connection interface. The following defines the class for the Daemon component.

```
class Cl {
        static Classification classification;

        public Cl();
        public String getClClass();
        public static void main();
}
```

The *classification* object will be discussed in greater detail in the following section. The *getClClass()* method is the remote method that is invoked by the web server. Its purpose is to return the client computing resource information in a string format. Even though most of the resource values are bound to be numerical values, Java provides methods that transform numbers into strings and vice-versa rather easily. The *main()* is responsible for collecting the client device connection speed to the Internet and providing this in the creation of the *classification* object. In addition, it provides the initialization required to set up the necessary RMI server to listen for incoming RMI requests. Once all the RMI initialization has been established the Daemon component will listen to port 1099 for incoming RMI requests.

3.1.2 Resource Component

The Resource component monitors the CPU utilization, memory utilization, and the type of processor. Depending on the operating system running on the hardware, each operating system provides different techniques to obtain this information, such as command line utilities, system library calls, etc. Thus, the implementation of the resource component is based on the Abstract Factory design pattern as detailed in [3]. The intent of the Abstract Factory pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes [3].

There are two classes to note. The first is the Classification class.

```
class Classification {
        String ClientClassification;
        ResourceFactory Factory;
        long waitinterval;

        public Classification (long);
      public String getClassification();
     public void Update();
}
```

The *ClientClassification* member is used to hold the client resource information in a string format that is obtained from the *ResourceFactory* object (which will be examined in detail shortly). The *Factory* member will contain an object for either the *SolarisFactory* or *WindowsNTFactory*. Lastly the *waitinterval* will contain a value to determine the duration at which the client resource information will be refreshed.

In the Classification constructor, we use a standard Java method call *System.getProperty("OS.name")* to get the operating system name. Based on that value, we are capable of determining which type of *ResourceFactory* object needs to be created. The *Factory* variable is initialized with this object. A call to *Factory.getCPUType()* is made to get the type of processor that the client has. This is used to determine the refresh rate at which the client polls for the client computing resource information (stored in the *waitinterval* member). On slower machines the update frequency is set to a longer interval and for faster machines to quicker refresh rates. The rationale behind this is that slower machines have fewer CPU cycles, so these cycles should be used more sparingly. Lastly the *ClientClassification* member is constructed to store client computing resource information as a string. The *getClassificaiton()* method is used to return the value of *ClientClassification* to the calling object (which will be the class being used to implement the Daemon process). Finally, the *Update()* method performs the operation of making method calls to the *Factory* object (e.g., *getCPUType()*, *getCPULoad()*, *getMemory()*) in order to get the current client computing resource information. The frequency of the updates is based on the *waitinterval* that was obtained in the constructor of the *Classification* object.

```
interface ResourceFactory {
String getDeviceType();
float getCPUType();
float getCPULoad();
float getMemory();
      }
```

The ResourceFactory is an abstract class. This allows for any number of subclasses to be created for any number of operating systems for various devices. Our prototype contains only a single implementation for the Solaris operating system. Future work will entail creating a similar ResourceFactory for the Windows operating system. With our implementation for the Solaris operating system, we were able to use native Solaris system calls to obtain the necessary information in the SolarisFactory. In obtaining the value for the getCPUType() method, we used the "**sysinfo**" utility to obtain the processor speed. For both the getCPULoad() and getMemory() methods, calls were made to the "**top**" utility for up to date utilization

data on the client device. In applying the Abstract Factory pattern, it would be relatively easy for new subclasses of ResourceFactory to be created and incorporated into the client component.

3.2 Server Component

Before detailing the server implementation for this paper, it must be recognized that portions of the Cocoon servlet from the Apache Software Foundation were used. The Cocoon software already provides the necessary infrastructure for handling incoming HTTP requests and transforming XML files with XSL stylesheets. We shall discuss only those components of the Cocoon software that we have made modifications to.

3.2.1 RequestHandler Component

The RequestHandler component is implemented using a Cocoon class implementation that extends the *HttpServlet* class, which provides the ability to handle HTTP requests and responses. Once the *Cocoon* object has been successfully created, it remains in an idle state waiting for incoming HTTP requests for a resource with the .XML extension. Upon receipt of such an HTTP request, the *service()* method of the *Cocoon* class extracts from the received request the resource that it is requesting. If the requested resource is a valid resource on the server, then both the HTTP request and HTTP response objects are passed to the *Engine* object (used to implement the QoS Component), otherwise an HTTP error response is sent back to client.

3.2.2 QoS Component

The QoS component is implemented as a class called *Engine*. The initialization of an *Engine* object requires from the *Cocoon* object information that includes the XML parser and the XSL processor to be used. These are used to initialize two objects that are encapsulated by the *Engine*: *parser* and *transformer*. It is apparent that only one instance of the parser and transformer objects will exist. Thus, the *Engine* object uses the Singleton pattern [3]. The intent of the Singleton pattern is to ensure a class only has one instance, and provides a global point of access to it. In our implementation, we use software from the Apache Software Foundation Thus, the XML parser is *Xerces* and the XSL processor is *Xalan.*

The *Handle()* method of the *Engine* object is the heart of the entire process. The Document Object Model (DOM) is constructed for the XML content. This is done through the *parser* object. The next step retrieves the individual data items of the client computing resource information (*QoS Data*) via a call to *Classification.GetClassData()* (the *Classification* object implements the Classification component and is discussed in the next section). The data that is returned is then passed on to the *Classifcation.VerifyClassData()* method that determines the *classification* value for the requesting client. The validity of the *classification* value is then determined with a call to *Classification.ValidClass()* and if the *classification* value is valid a call to *Classification.ReturnLocation()* for the directory location of the XSL stylesheet is made. Based on the number of classifications, there could be any number of directories for stylesheets. In this example, we assume three classifications for constructing simple web pages, default web pages, and complex web pages. Once the location of the XSL stylesheet is determined, the actual XSL stylesheet is applied

to the XML DOM using the XSL processor. Once this has completed, the processed contents from the processor object are inserted into the *response* object, which is then delivered to the client device to be displayed.

3.2.3 Classification Component

The Classification component encapsulates that information required to assign a *classification* value to clients based on their computing resources and a mapping from *classification* values to a presentation descriptor value. This component is implemented using the *Classification* class. The initialization of the *Classification* class retrieves from files the information needed to classify the clients.

```
public class Classification implements Defaults {

String[] classdata;
int ClientClassification;
float CPUTypeWeight;
float ConnectionWeight;
float CPULoadWeight;
float MemoryWeight;
float[] ClientClass;
float[] CPUTypeClass;
float[] ConnectionClass;
float[] CPULoadClass;
float[] MemoryClass;

public Classification();
private float LoadWeight(BufferedReader infile);
private float[] LoadClass(BufferedReader infile);
public String GetClassData (HttpServletRequest request);
public int VerifyClassData (String value);
private float CalculateWeight(float resourcevalue, float[] inputclass, float weightclass, int flag);
private int DetermineClass (float clientvalue, float[] inputclass);
public String ValidClass (int classdata);
public String ReturnLocation (int locationclass);
}
```

The *Classification()* method (used to create the *Classification* object) uses the private *LoadWeight()* method to load from a file the weight values into the following variables: *CPUTypeWeight*, *ConnectionWeight*, *CPULoadWeight* and *MemoryWeight*. Table 1 provides an example set of weights.

Table 1. Resource Weights

Member	Value
`CPUTypeWeight`	30
`ConnectionWeight`	30
`CPULoadWeight`	30
`MemoryWeight`	10

The total combined weights must sum to 100. Each resource can be considered as a percentage, providing an administrator with an easy way to judge the significance of each resource.

The private method *LoadClass()* loads the variables *CPUTypeClass*, *ConnectionClass*, *CPULoadClass* and *MemoryClass.* These variables are arrays.

Each array stores measurement values. A measurement in an array location *i* represents a boundary point. Thus, for any array *a*, (*a[i],a[i+1]*) represents a categorization of values. All values less than *a[0]* is a categorization and all values greater than *a[n-1*] is a categorization, where *n* is the number of categories , is a category. These values are used to determine classifications. Table 2 shows an example.

Table 2. Resource Categories

Member	Array Location			
	0	1	2	3
`CPUTypeClass`	90	166	300	450
`ConnectionClass`	28800	56000	500000	1500000
`CPULoadClass`	75	50	25	XXXXX
`MemoryClass`	66	33	XXX	XXXXX

Not all of the arrays will contain the same number of entries. In this example, the *CPUTypeClass* member has four entries and the *MemoryClass* contains two entries. The *CPUTypeClass* is used to categorize the speed of the process in the client device. As can be seen, the categories correspond to devices with less than a 90 MHz processor, 90 MHz to less than 166 MHz, 166 MHz to less than 300 MHz, 300 MHz to less than 450 MHz, and over 450 MHz. We note that the *CPUTypeClass* and *ConnectionClass* are stored in ascending order, while *CPULoadClass* and *MemoryClass* are in descending order. The reasoning is that in the case of the CPU type and Connection speed, a higher value means that there are more computing resources and therefore, the system can more quickly respond to content from the server. With the CPU load and Memory load, a higher value represents degradation in system performance.

The initialization process also constructs an array that maps a classification level to the associated directories for the XSL files. Table 3 provides an example.

Table 3. Classification Categories

Member	Array Location		
	0	1	2
`Classdata`	/simple/	/default/	/complex/

The method *VerifyClassData()* is used to verify that the contents that were returned by the *GetClassData()* method is properly formatted. If the string is properly formatted, the calculation to determine the classification of the client begins. We will use the following string in our example: *0.1 COMP 269 56000 30 40*. The first thing checked is the version number of the incoming information. This will be important if future work determines that other client computing resources are found to influence the performance of a system. The second item to be checked is the type of device making the request. If the device is other than a computer running a web browser, different computations may be required. In our work, we have accounted for the standard computer and web browser with "COMP". However, another possible device is that of a wireless cell phone, in which case we will have the value of “WLESS”. This will require that we provide another directory for “/wireless/” that stores a unique XSL stylesheet to transform the content to be displayed on a cell phone type device using WML. In this example, we are dealing with a computer and

web browser. The next piece of information is the type of processor in the computer, which is 269 MHz. A call is made to *CalculateWeight()*. The value 269, the *CPUTypeClass* and *CPUClass* arrays and a flag value of 1 are passed to *CalculateWeight()*. If the flag value is 1 then the array passed in is in ascending order otherwise it is in descending order.

The *CalculateWeight()* method is aware that the array *CPUTypeClass* is an array of ascending values since the flag value is 1. At this point, it checks each value at location *CPUTypeClass[i]* until it finds a value of *CPUTypeClass[i]* that is greater than 269. In this example (based on Table 2) this is the third array location (and thus the third category). There are five categories in which the CPU types are separated (these include under 90 and over 450 MhZ). In our implementation the category value is computed based on the array location and is equal to $(i+1)/n$, where i is the category and n is the number of categories. In our example, since value 269 is in *CPUTypeClass[2*], the category value is 3/5. The weight for the CPU type was defined to be a value of 30, thus we arrived at a calculation 3/5 * 30 = 18 for the CPU type in our example. This value is returned to the *VerifyClass()* method and a running tally is kept, as the *CalculateWeight()* method is called three more times, once for each of the other resources of Connection speed, CPU Load, and Memory Load.

For connection speed the calculation is 3/5 * 30 = 18. This value is returned to the *VerifyClass()* method. The total grows from a value of 18 (from the CPU Type) to 36.

The next value to be calculated is that of 30 for the CPU Load. The array, *CPULoadClass*, is organized in descending order (as indicated by a flag value of –1). Instead of comparing to see if the CPU load passed in is less than CPULoadClass[i], the comparsion is a greater than or equal to. Since the CPU load falls into the third category, we obtain a calculation of 3/4 * 30 = 22.5. Once back into the *VerifyClass()* method, the total value grows from 36 to 58.5.

Once again we must perform the same calculations on the Memory Load. In performing the calculations 2/3 * 10 = 6.6666, we get a value of roughly 6.667. Once back in the *VerifyClass()* method, the total value grows from 58.5 to a value of 65.167.

In working through this example, it is apparent that the highest possible value that can be calculated in this process is 100. Every request that is received with a ".xml" extension will be processed and assigned a value ranging from 0 to 100. This design allows us to easily extend any of the resource categories, as well as adding additional resources at a future time without affecting the number of classification categories stored.

We have now calculated a value for the resources of the client computer to be that of 65.167. Still in the *VerifyClass()* method, we must determine which *classification* value is assigned to the client request. This is done by making a call to the *DetermineClass()* method. In a similar manner as above, the *ClientClass* is an array with boundary values at each of the array locations. The values of 25 and 60 are the entries in the array locations for 0 and 1 respectively. The value of 65.167 fits into the third category. In this method, the counting of categories begins at 0, so the first category would be a value of 0, the second category a value of 1, and the third category of 2. A value of 2 is returned to the *VerifyClass()* method. As this is the last item performed by the *VerifyClass()* method, this same value is returned to the object that had called *VerifyClass()*, in this case a value of 2 is returned. In this example, we were able to calculate a valid classification for the request. There are instances where an error message would be returned in place of a valid classification. For example, if

during transfer the client resource information were to become corrupt and instead of receiving a numeric value for the CPU type, the server received a string of characters. The processing of the request would fail and an error code would be returned instead of a valid classification.

The *ValidClass()* method would be called after the *VerifyClass()* method to determine the nature of the validity for the calculations and the returned value. In examining the value, it is determined if it fits into a classification or if the value were an error code. If the value is an error code, the appropriate error message is constructed and an error string is returned from the method. Otherwise, a null string is returned if the value is a valid classification.

The last method for this object is the *ReturnLocation()* method, which returns the directory location for retrieval of the appropriate XSL stylesheet to be applied.

4 Experiments

This section briefly describes our initial experimentation.

4.1 Experimental Environment

The experiments were performed on an isolated network with three Sun Ultra workstations. The workstations varied in their hardware configurations. Two of the machines functioned as the clients that requested web content via HTTP. These machines are named Strawberry and Doublefudge. Strawberry is a Sun Ultra 5/10, with a 269MHz UltraSPARC-IIi processor and 128MB of physical memory. Doublefudge is a Sun Ultra 60, with dual 359MHz UltraSPARC-II processors and 1024 MB of physical memory. While the third workstation named Vanilla, a Sun Ultra 5/10, with a 269MHz UltraSPARC-IIi processor and 256MB of physical memory, was used as the host delivering the web content.

It was critical in our experimentation to simulate connection speeds equivalent to those most commonly used today. This was achieved by establishing a Point-to-Point Protocol (PPP) connection via a serial interface. In establishing a PPP connection we were able to control the connection speeds between the two client machines of Strawberry and Doublefudge to the host machine Vanilla. The maximum connection speed that was achieved was equivalent to 400 Kbps, while using the lower extreme of 38.4 Kbps.

The web server used in our experimentation was Java Web Server 2.0. This web server software provided the necessary infrastructure to handle servlet execution without requiring additional components. In addition to the Cocoon 1.6 Java servlet from the Apache Software Foundation, we also required an XML parser and XSL processor. These two were also obtained from Apache, with the XML parser being Xerces 1.01 and the XSL processor being Xalan 19.2.

As mentioned earlier, it was necessary to produce heavy CPU load and memory load situations on the client workstations. The added load on the clients was achieved by the use of two utilities that we developed. The first is a CPU load generator utility and the second is a memory load generator. These two programs allowed us to generate high and low load values for both CPU and memory usage.

We also developed a utility that allows the user to make specific HTTP requests for a specific URL from a web server at the command line. In using this utility, scripts were developed to eliminate the need for manual requests through a web browser. In using this utility, our experimental results are based on the HTTP 1.0 design, in which HTTP connections do not remain persistent, instead each object in the web page requires a completely new connection to be established.

4.2 Factor Analysis for Original System

The first set of experiments performed were used to determine which of the four factors: Connection speed, CPU type, CPU load, and Memory load contributed most significantly to the delivery of web content to the client device. We used a 2^4 full factorial design with 3 replications. Each of the factors had a high and low level. Four experiments were performed, in a similar manner. The first experiment consisted of the web server delivering standard HTML web content, with a default page having contents equaling 28KB. The results from this experiment found that the factor contributing the most to the content delivery was the connection speed, accounting for over 69.747 and the CPU load for 14.815 of significance. A similar experiment was performed with a complex HTML page equaling 1500 KB. This time the connection speed accounted for almost all of the significance with a value of 99.904 and all others were insignificant.

We then performed the experiment with the same factors, but instead making requests for the XML content through the original Cocoon servlet. The default XML page, being the same size as the HTML content of 28KB, produced almost identical results as the HTML default page. The significant values were once again the connection speed with 66.033 and the CPU load at 15.919. Once again with the complex XML content, the connection speed was most significant with a value of 99.849 and all others being insignificant.

In studying these two scenarios, the HTML content delivered by the web server is similar to a file server, obtaining stored files from a storage media and delivering it to the client. While the XML content requires the web server to process the content by applying an XSL stylesheet to the XML content. It appears with our experimentation that this additional overhead of processing only increases the response time marginally.

4.3 Prototype Results

From examining the results for the factorial experiments, the most significant factor for “key to glass” response time for the client device is the connection speed to the Internet. Based on this, we assigned the greatest weight to the connection speed of 70%, followed by CPU load at 20%, and the Memory load and type of CPU each with 5%.

A time of over eight minutes was required to obtain the 1500KB complex web page from the web server with a slow connection in our previous experiments. With knowledge of the slow connection being the most significant factor, our prototype will only provide the complex content if the user is on a fast connection. Hence eliminating the eight minutes of wait time for the complex web content.

The comparison of the times is taken with respect to the original Cocoon implementation. Included with this timing is the duration required for the Java RMI call to the client to obtain the QoS data, based on the *pull* model. In our QoS enabled servlet implementation, we also provided a timing for the RMI call. So, we were able to record the exact time required to perform the remote call. On average the RMI call from the web server to the client took 432.014 milliseconds to complete. This value is rather small in nature and didn't contribute greatly to the overall "key to glass" response time.

5 Related Work

5.1 Replicated Servers

When examining the idea of replicated servers on the Internet, we are interested in two key characteristics to help improve "key to glass" response time. The first factor is the distance to which the web server is located from the client requesting the web content. In saying distance, we can not assume the physical locality of the web server, but more importantly we must use the network hop distance. This value represents the distance needed to travel to arrive at the web server by the client request over the Internet. It is evident if the request and response spends a large amount of time travelling on the network, the response time experienced by the user will be affected.

The second crucial factor is the time required by the web server to process the client request and ultimately deliver the appropriate response. When considering this factor we must also examine additional items that affect the web server's ability to process requests. One such item deals with the number of requests coming into the web server, as from 1998 to 2002 the number of users on the WWW are expected to triple in volume [5]. It is apparent that a web server could potentially become over burdened serving client requests. As well as the client requests, the other item of great significance is the type of requests the clients are submitting. At the onset, web servers delivering content were very similar to file servers. A client request would arrive at the web server and the web server would deliver static web content to the user. There was very little computational overhead associated with client requests. With the evolution of the WWW, web content complexity grew hand in hand. In addition to static web content, there are now web sites that dynamically create web content unique to each client request, providing a personalized experience. In these dynamic web pages, they are not just accessing static content from files, but they are accessing databases to obtain client information, as well as other web sites for up to date content. The technology behind the dynamic content creation has also advanced, with traditional Common Gateway Interface (CGI) scripts and applications, Active Server Pages (ASP) from Microsoft, Java servlets, to eXtensible Markup Language (XML) and eXtensible Style Language Transformation (XSLT). Along with these technologies comes the added processing required by the web server. Web servers are becoming more complex entities, they are no longer just file servers, but have the added responsibility of creating dynamic web content.

5.2 Caching

The idea of caching is similar to that of Replicated Servers, which was discussed in the previous section. The intent of caching is not to replicate the entire server's contents, but to only reproduce those portions that have been frequently accessed. In doing so, the load of the web server will be reduced.

In examining caching, there are three points at which the caching can take place. To examine these three points, let us begin by looking at the client making requests for web content. This is the very first place that caching can occur. Once a client makes a request for web content the web content can be cached at the local client. In almost all instances, the caching is the responsibility of the web browser, as the browser is responsible for the handling of requests and the returned responses. If the browser notices that a request for a particular web resource has been cached, it will retrieve it from the cache, and bypass the submission of the request to the web server. This helps in improving response time by providing the obvious advantages, first there will be no network latency as the request doesn't even need to travel onto the Internet. Secondly, the web server will not need to service the client request, hence eliminating the server load. The advantages have been shown and the disadvantages will be discussed later in this section.

The next point for caching to occur is at a proxy location. Proxies can reside at various locations, but we will refer to the location at which an autonomous system is connected to the Internet. Since all the client requests on the autonomous system need to travel through the proxy to reach the Internet, it is ideal for this proxy to cache web content responses. To understand the sequence of events that occurs in this situation, we will have two clients *C1* and *C2* on an autonomous system that travels through proxy *P*. *C1* begins by making a request for www.needinfo.com, the request travels to *P* and then travels out onto the Internet to the www.needinfo.com site. The response is returned by the web server hosting www.needinfo.com and the response travels to *P* where the response is cached. *P* then sends the response to *C1* where it is processed. Later that day *C2* also makes a request to www.needinfo.com, the request travels to *P*, but the response for *C1* was previously cached, *P* needs only to return the cached contents to *C2*. In this manner, the *C2* request only needs to travel to *P* and doesn't even need to travel onto the Internet to the www.needinfo.com web server. Similar to caching at the client only, this approach helps both reduce the amount of traffic travelling onto the Internet and reduces the server load by eliminating repeated requests.

The last point of caching can take place directly at the web server that is handling client requests. The rationale behind this caching is to eliminate the processing load from the web server. For those instances of complex web content, in which server processing is required. Instead of performing the processing for each client request, it would be beneficial to cache the web content that is produced. The client request would arrive at the web server, it would look through the cache for the web content, if it exists then it would be retrieved, otherwise the web server would process the request to deliver the web content. Even though this process doesn't alleviate the network traffic to the web server, it helps reduce the server load by eliminating repetitive processing of identical requests.

As shown through the three caching locations, caching is an advantageous technique to improve upon response time for the user. Along with the benefits there are also definite drawbacks to this technology. The first to come to mind is the need

for cache consistency. Cache consistency is the requirement to keep the cache content, either being at the client cache, proxy cache, or server cache, consistent with the content that is delivered by the web server. In any of these instances, web content may be cached in these three locations, in addition there may be updates at the web server delivering the content. The cached locations must be aware of this update, so that out dated content is not delivered to the requesting client.

The second problem deals with personalization on the WWW. This deals with web sites that personalize web content based on the individual, for instance a web site that greets you by your first name in the web content that is returned. This could happen at a banking site, which requires you to log in, once logged in the web site is aware of who you are and creates content specific to you. When using caching, this personalized content is not relevant to others accessing the same web site, as the web server would create content that is specialized for that individual. In this manner the content is created dynamically, there are also other instances in which dynamic content could be created but not personalized. An example could be an e-tailer that creates web content for merchandise they sell on the WWW. This is dynamically created based on the price of merchandise that is stored in another database, but is infrequently changed. In this instance, rather than have each request to the web server processed, this content can indeed be cached. In looking at the situation from this standpoint it would be beneficial for dynamic content caching. Again we must weigh the advantages gained by caching the dynamic content and the possible problems posed if the cached contents become outdated. In this manner not all web content can and should be cached to improve response time.

5.3 Adaptive Web Content

At the beginning of this paper we stated the significance for “key to glass” response time. In the previous two sections we looked at the techniques of replicated servers and caching in an attempt to stay within a 10 seconds threshold. In examining these other two techniques you will notice some similarities. The first is the desire to alleviate work imposed on the web server, which is delivering the content to the client. With replicated servers, the workload is distributed among several identical servers. On the other hand with caching, attempts have been made to cache copies of the dynamically created content, which totally eliminate the processing required on the web server.

The second common characteristic is the desire to bring the content closer to the requesting client. Once again with replicated servers, the ideal situation is to strategically place identical servers at various locations on the Internet. In a similar manner with caching, if there are strategically placed cache servers, the time traveled on the network will decrease, along with the decreased time traveled on the network there will be an increase in the “key to glass” response time.

You are more than likely pondering the question, why do we need adaptive web content when replicated servers and caching appear to improve the “key to glass” response time. The techniques suggested in the previous two sections are possible solutions, but with the solutions exist drawbacks, especially with the present state of the WWW. To improve response time in both techniques requires additional servers to be placed throughout the Internet. For companies and more importantly

individuals, who are restricted by limited budgets cannot afford the cost of additional servers and network connections.

As the WWW grows, the trend for web content is towards dynamic personalized content rather than the more traditional static content that is seen today. This is due in part to the increased number of retailers providing e-commerce on the WWW. The information for the e-tailers is usually stored in a database, with dynamically created content containing queried data from the database. For example, an e-tailer may sell widgets on the WWW, they already have a legacy database system that contains all the prices of the widgets they manufacture. The price of these widgets changes monthly, and the database has several thousand widgets. Instead of creating a new set of web content containing the updated prices monthly, the e-tailer instead queries the existing database for the price quotes. Hence, the only way to create the dynamic content for a price quote is for the web server to dynamically create the content for each client request.

In addition to dynamic web content, the amount of multimedia content on the WWW is also increasing. The e-tailer who is selling widgets on the WWW today is not only giving a descriptive commentary of the widgets, but is also providing graphical images of each widget they sell. The number of images and resolution for each image are also increasing. With the increased number of people accessing the WWW, the size and complexity of web content is also rising. This is where adaptive web content will benefit the client's "key to glass" response time. In an attempt to decrease the response time, web servers perform two actions when processing a client request. The first item is the actual creation of the content to be returned to the client, this includes obtaining the content to place into a web page, such items as retrieving the links to image files from disk, making the database queries, and formatting the content appropriately. The second such task is the actual retrieval of static data from hard disk and delivery of the content to the client. Of the two processes, most of the research has concentrated on the second step concerning the retrieval of static data and the delivery of that content to the client. The interest of this technique is not to reduce the total amount of content delivered, but to only reduce content when the server is not capable of delivering a level of QoS to the client, and at that point reduce the content to once again achieve the expected level of QoS.

6 Conclusion

As the number of users accessing the Internet and WWW increase, it is expected by the year 2002 there will be over 320 million people accessing the WWW alone [5]. That is why we see the influx of e-tailers to the WWW. In addition, the demands of the users are also increasing, with the most predominant being the QoS expectation, in particular with the response time for the delivery of web content. As mentioned earlier this is crucial for people returning to a web site, as long delays can guarantee loss of customers.

Presently, with most web servers there is no consideration for the resources of the client. All client requests are treated as if they were equal in nature, even if the client hardware and network connections differ greatly. Our research supports the conclusion that others have shown in that the client resources, in particular the Internet connection, can greatly affect the response time experienced by the end user.

In our implementation we were able to show that providing adaptive web content can greatly improve upon the user's QoS.

Present web sites must cater to the lowest common denominator, which are clients connecting to the Internet with a phone modem at 28.8 Kbps. This is done to make sure all the visitors to the web site are content with a certain level of QoS, especially those with the slow connection. What about the users on fast connections? This doesn't allow them to take full advantage of their connection. With our design, the web site provider has the ability to create various levels of content based on the available client resources. In using this technique, an e-tailer's web site could provide a short video clip to the user on the fast connection, but provide a single jpeg image to the user on the slow connection for the merchandise they are trying to sell. In both circumstances the users will receive the information on the merchandise. However, the e-tailer does not have to store different versions of the content. Hence, they save on disk storage and maintenance costs. Although, XSL style sheets are to be stored, these will not have the same storage costs of storing different versions of the content.

Further work includes developing the push model, to have each client HTTP request include the QoS Data. We will develop various Resource Factories for different operating systems, in particular those for Windows and Linux operating systems. There could also be additional client computing resources that affect the response time. Further experimentation will examine this as well as determine this. We will also examine the problem of optimal categorization and weights. Finally, we must further examine the effectiveness of this approach from a user's point of view. Currently, we only look at client computing resources. We would also like to include user profiles that allow us to also use priorities.

Acknowledgements

This work is supported by the National Sciences and Engineering Council (NSERC) of Canada, the IBM Centre for Advanced Studies in Toronto, Canada and the Canadian Institutute of Telecommunications Research (CITR).

References

1. Abdelzaher, T. F., Bhatti, N., Web Content Adaptation to Improve Server Overload Behavior, In *Proc. of the 8th International World Wide Web Conference*, Toronto, Canada, May 1999.
2. Bouch, A., Kuchinsky, A., Bhatti, N., Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service, *Internet Systems and Applications Laboratory, HP Laboratories*, Palo Alto, January, 2000.
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Inc., 1997.
4. Seshan, S., Stemm, M., Katz, R. H., Benefits of Transparent Content Negotiation in HTTP, *Proc. Global Internet Mini Conference at Globecom 98*, Sydney, Australia, November, 1998.
5. "The Internet, Technology 1999, Analysis and Forecast", *IEEE Spectrum*, January, 1999.

Discussion

F. Paterno: The most important portion of your QoS data seems to be the static data. Is it true that only static information is necessary.
H. Lutfiyya: That seems to be the case right now but I have a hard time believing that this will hold through future experiments.

F. Paterno: Key to glass time is not a single number but is spread out over time. Have you thought about capitalizing on that.
H. Lutfiyya: We have not yet considered that.

D. Salber: The RMI call takes 432 milliseconds. Is that per HTTP request?
H. Lutfiyya: That is per HTTP request. That is using the pull model and if we used the push model the latency should be reduced.

S. Greenberg: I would have thought that this could be done more efficiently at the server side by the content designer. That is, embed the sequence into the page a priori rather than deciding this through inference.
H. Lutfiyya: Right but that might be somewhat slower. You might be slowing things down for your high speed users.

J. Roth: Why do you choose something like CPU load and memory load since these factors change quickly over time. What have no categories related to the graphical capabilities of the client.
H. Lutfiyya: I agree with you. There are other hardware characteristics that we should incorporate. We just haven't done it yet.

L. Bass: Often web page responses are spread out over time with the page filling in slowly. Have you thought about capitalizing on this.
H. Lutfiyya: No