JavaCGIBridge: A Java and HTTP Communications Framework

Introduction

Due to a combination of a stringent security model and the relatively short time the Java language has existed, applets have limited options to communicate with other processes and databases on a distributed network. CORBA, RMI, JDBC, or proprietary socket communications tend to either be highly browser dependent, have less than optimal communications/library size overhead, or run into security obstacles such as firewall compatibility.

One alternative to these heavy communications layer is a simple Applet to HTTP/CGI script communications framework. The JavaCGIBridge framework was written with the purpose of providing this alternative in a programmer-friendly way.

Problems with Traditional Client-Server Technologies

JDBC (Java DataBase Connectivity), while an JavaSoft standard, is not a protocol that is efficient for most applet projects. First, commercial JDBC driver implementations tend to be bloated in both size (upwards of 200k) and sheer number of classes to download. Figure 1 shows a typical JDBC applet architecture.

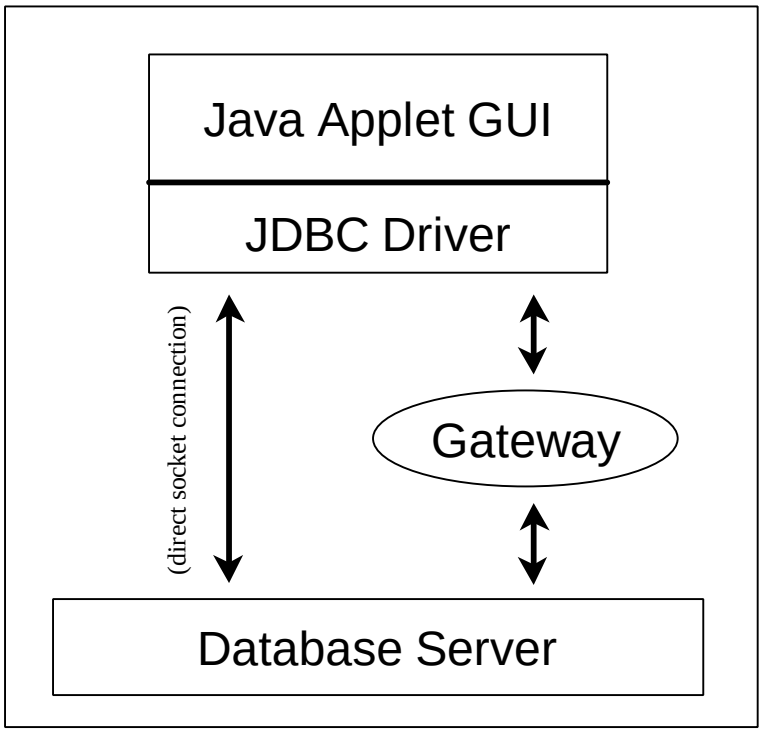

Figure 1 – JDBC Applet Architecture

Second, many vendors do not support the old JDK 1.0.2 version of JDBC so the browser audience is limited. Of course, this may be just as well given that the 1.0.2 version would require a browser to download the entire jdbc.sql.* hierarchy of classes on top of the database specific JDBC driver itself.

Third, coding SQL logic directly in the applet tends to result in larger applet sizes as the business rules tend to migrate to the applet itself since JDBC communicates directly with the database server. The exception to this is that Sybase's Jaguar Transaction middleware understands JDBC. However, Sybase's JDBC driver still suffers from the first two problems even with this exception.

Finally, there are potential security problems using JDBC. Many JDBC drivers do not understand SSL (secure socket layer) or other encryption protocols. Thus, passwords and database data are at risk of being sent in the clear to the database server. There are, however, some JDBC drivers that use SSL tunneling and Web server middleware to emulate security, but your database may not support those features.

For example, Sybase's JDBC driver presents a tradeoff to applet developers. The first release of their JDBC driver is compatible with JDK 1.0.2 browsers but lacks SSL tunneling. Their latest release support SSL tunneling for security, but along the way Sybase decided to drop JDK 1.0.2 compatibility. Thus, you can have security but limit the browser, or allow your Sybase JDBC applet to be seen on all browsers but have limited security.

Object-based middleware such as CORBA (Common Object Request Broker Architecture) and RMI (Remote Method Invocation) also have problems. RMI is not compatible with all browsers. CORBA classes, like JDBC classes, tend to be bloated in size and number of classes. Relying on browsers with CORBA libraries built into them such as Netscape 4.x, limits your browser audience. Proprietary socket based middleware such as ObjectSpace Voyager have similar issues to the CORBA/RMI object middleware.

Using JavaCGIBridge to Mitigate Traditional Client-Server Issues

The only truly open protocol that all Java-capable browsers understand is HTTP. In addition, Java applets within Browsers natively understand how to use SSL communications, rendering communications secure if the programmer sets up SSL connections.

As an added bonus, marrying applets to CGI scripts or Java Servlets actually makes a great deal of architectural sense for a Web Application. Client-side Java has long been considered a difficult undertaking especially with subtle GUI quirks on various platforms.

You can limit this risk but coding the majority of your application in a traditional HTML form-based application using CGI scripts or Servlets. This still allows you to leverage Java applets where user friendly or unique data visualization GUIs enhance the experience for Java capable users. In other words, Java

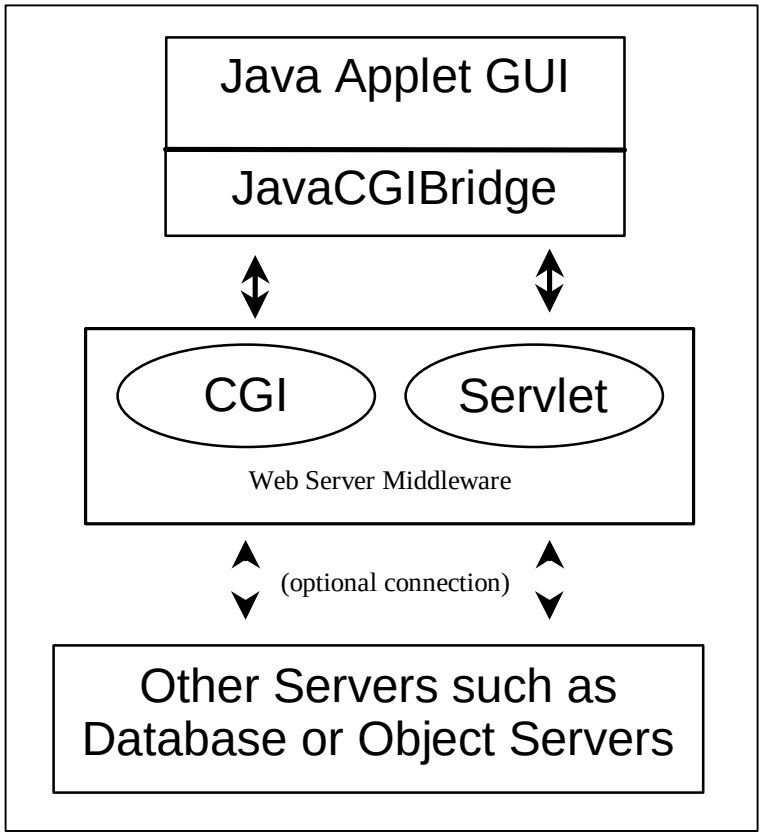

Figure 2 - JavaCGIBridge Approach to Applet Design

applets are only used where they are necessary—the rest of the web application can remain using HTML forms based user-interface.

By using HTTP protocol to communicate with the web server, an applet can leverage the code that a programmer has already written for the business logic side of the HTML forms interface. Put another way, "legacy" CGI code written in a language such as Perl can be leveraged without totally throwing it away simply because a new type of data display mechanism such as Java is being used.

The JavaCGIBridge class library framework was designed to provide a solution to these problems. It allows the casual programmer to code an applet that talks to a Web server without a huge learning curve. Plus, it has sophisticated features allowing a systems architect to take advantage of useful but more difficult to implement features such as serialization of Java Objects or asynchronous communications requests. Figure 2 shows the JavaCGIBridge framework relationship.

A First Look at JavaCGIBridge Code

The core JavaCGIBridge framework consists of a class library that posts information to a CGI script and can read the information back in. If the information is returned with programmer-specified delimiters, the framework can automatically parse the data into a Vector of Vectors. Each element of the first Vector contains Vectors that represent each record that was retrieved. Each record Vector contains elements that contain Strings representing each field in the record. Figure 3 shows an example of this structure if six records were returned and four fields were sent as part of those records.

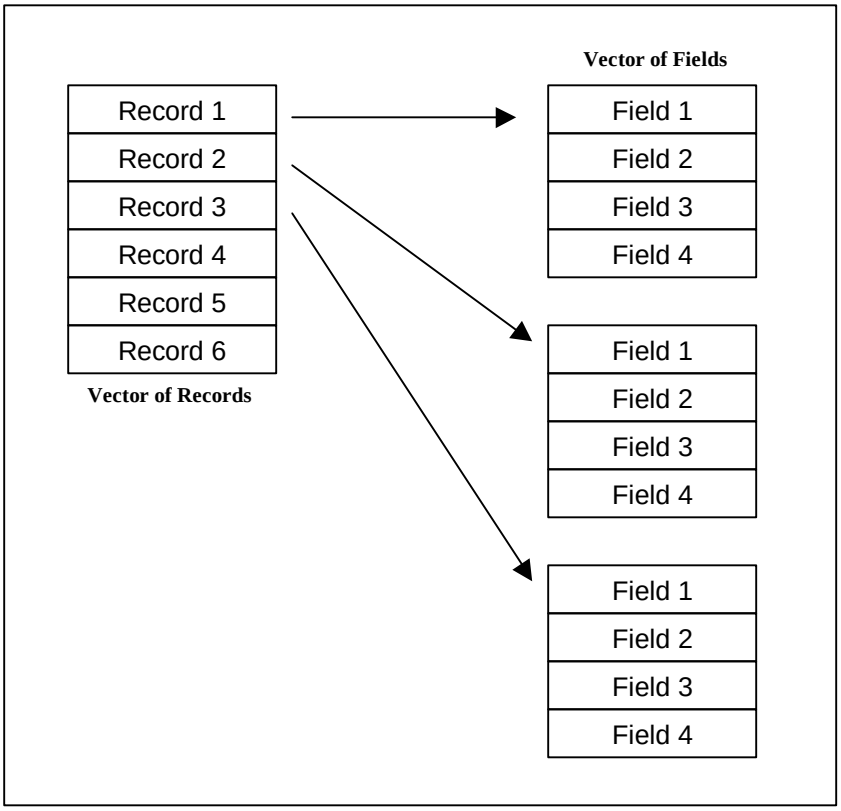

Figure 3 - Vector of Vectors Parsed Data

The following code shows a simple example of using the JavaCGIBridge to post "firstname" and "lastname" variables to an "addressbook.cgi" CGI script which presumably returns addressbook information such as phone numbers and locations for the queried first and last names. The myURL variable contains a reference to "addressbook.cgi" for the purposes of this example.

The formVars Hashtable is the object which associates form values with Hashtable keys that represent the input form variable names. In the example below, “lastname” would be a key to the Hashtable and “Birznieks” would be the value assigned to that key. Although I am only associating one value with a key in this particular example, JavaCGIBridge handles allowing multiple values to be assigned to a key.

```
JavaCGIBridge jcb = new JavaCGIBridge();
Vector returnedDataSet = null;
Hashtable formVars = new Hashtable();

try {
    jcb.addFormValue(formVars, “firstname”, “Gunther”);
    jcb.addFormValue(formVars, “lastname”, “Birznieks”);
    returnedDataSet = jcb.getParsedData(myURL,formVars);
} catch (JavaCGIBridgeTimeOutException e) {
    System.out.println(“JavaCGIBridge Timed Out:” + e);
}
```

Since JavaCGIBridge uses a pipe symbol (|) for a default field delimiter and a newline (\n) as a default record delimiter, using the getParsedData() method will parse the returned raw data into the returnedResultSet Vector. Then, you can use some code to enumerate over the returned Vector of Vectors. Sample code that would print out the results as comma-delimited fields to System.out is shown below.

```
for (Enumeration e = returnedDataSet.elements(); e.hasMoreElements();) {
    Vector v = (Vector)e.nextElement();
    String s = “”;
    for (Enumeration innerE = v.elements(); innerE.hasMoreElements();) {
        if (s.length() > 0)
            s = s + “,”;
        s = s + (String)(innerE.nextElement());
    }
    System.out.println(s);
}
```

The above two code snippets demonstrate how few lines of code it takes to get data from a web server via an applet using JavaCGIBridge. Of course, parsing records is probably the minimum that an applet may want to do in order to retrieve data from a CGI script or a servlet. The JavaCGIBridge framework also supports a host of other features in order to aid communications. These are discussed in the subsequent architecture and design sections. In addition, examples of all the major features are provided at http://gunther.web66.com/JavaCGIBridge/.

General Architecture

The general architecture of the JavaCGIBridge framework focuses on three major design constraints. While there were other governing factors to the design such as common Java idioms, the following design constraints represent the foundation upon which the JavaCGIBridge framework was built.

The first constraint is that the library should be simple. Second, the core classes should remain as small as possible. Finally, the class library should be flexible enough to handle advanced features such as object serialization without adding bulk to the core library. In other words, programmers should only pay for features they use.

These three factors in the JavaCGIBridge architecture actually worked hand-in-hand during development. By satisfying the first constraint by keeping the core classes small, I also ended up limited the API so that programmers who wanted to do simple things did not have to wade through a lot of extra documentation. This resulted in satisfying the second constraint.

In addition, to keep the core classes small, I partitioned advanced features into other classes. This resulted in a design where a programmer did not have to download the additional class if they did not wish to use the less common or more advanced features. Had I kept all the features in the core class, I would have been faced with either a very large and monolithic core class library or I would have had to drop some features.

The entire JavaCGIBridge framework consists of six classes. Table 1 shows which classes are included. In addition, core classes are labeled. Core classes are considered to be classes that are required by every JavaCGIBridge program. The other classes are optional depending on the features you wish to invoke.

In satisfying the constraint that the core classes be kept small, the core JavaCGIBridge class takes up 8.6k. Even all the classes together take up less than 20k. This is less than ten percent of the size of some of the commercial JDBC and Object Middleware class libraries that exist!

Table 1 – JavaCGIBridge Framework Classes

Class	Major Features	Core	Size
JavaCGIBridge	Uses POST method to send data to CGI Retrieves data as raw byte array Retrieves data parsed in Vector of Vectors Retrieves setup data as a Hashtable Sets up communications timeouts Observes data as it is retrieved or parsed Can return immediately after POSTing data to a script using the callOneWay() method	Y	8.6k
JavaCGIBridgeTimeOutException	Thrown if the class times out Sent to Observer to inform of timeout	Y	.3k
JavaCGIBridgeExtension	Inherits from JavaCGIBridge Adds methods to send Serialized Objects Adds methods to retrieve Serialized Objects Call CGI scripts with showDocument() Retrieve using fetchNextRecord() method	N	3.3k
JavaCGIBridgePool	Manages JavaCGIBridge objects Sets up default JavaCGIBridge values Can set maximum objects in use Can run low-level clean up thread Can clean up objects manually	N	4.4k
JavaCGIBridgeNotify	Contains notification info sent to Observers through the update() method	N	.8k
JavaCGIBridgeStillProcessingException	Thrown if a programmer tries to call a JavaCGIBridge object method while it is still processing information.	N	.3k

Core JavaCGIBridge Design

The main core class is JavaCGIBridge. This class contains the communications code which allows data to be posted to a CGI script, raw HTML data to be returned, and that same raw content to be parsed if it is in a format that the programmer specifies such as comma-delimited fields.

Although I wanted to keep the core class file size down, I did end up adding several additional interfaces to these core features. These include being observable by objects that implement the java.util.Observer interface, performing asynchronous calls to CGI scripts, and reading setup file information. These extra features were generally added to the core class because they integrated so tightly with the core parsing code.

Table 2 contains a summary of the methods that can be called from a core JavaCGIBridge object. The methods are organized by category. Data retrieval methods interface with CGI scripts and servlets. Property methods change the behavior of the JavaCGIBridge object. And finally, helper methods exist to perform commonly used functions. The specifics behind what the methods do are discussed in the sections below.

Table 2 – JavaCGIBridge Public Methods

Category	Method Name	Function
Data Retrieval	getParsedData	Retrieves CGI output in parsed form
Data Retrieval	getRawData	Retrieves CGI output as raw HTML in a byte array
Data Retrieval	callOneWay	Calls a CGI script but returns immediately
Property	get/setParseAsRaw	The default is false so CGI output is parsed as Java Strings. Setting this to true parses each field as a raw byte array.
Property	get/setParsedNotifyInterval	Determines how often Observers get updated when more records have been parsed
Property	get/setRawNotifyInterval	Determines how often Observers get updated when more raw bytes have been received
Property	get/setStatus	Allows you to see the processing status
Property	get/setTimeOut	Get and set the timeout threshold
Property	get/setStartDataSeparator	Get and set the start of data separator for parsing
Property	get/setEndDataSeparator	Get and set the end of data separator for parsing
Property	get/setFieldSeparator	Get and set the field separator for parsing
Property	get/setRecordSeparator	Get and set the record separator for parsing
Helper	getURLEncodedHashtable	Take a hashtable with form variable/value pairs and create a URLEncoded string out of it for POSTing and GETing CGI scripts.
Helper	addFormValue	Adds a form variable/value pair to a Hashtable for eventual POSTing to a CGI script.

Retrieving Data

The core JavaCGIBridge class supports three main methods for retrieving data. The first two methods return data directly to the caller and so the program will wait until the method returns. The third method simply runs the URL and returns control to the program immediately.

getParsedData() returns a Vector of Vectors which have been parsed from the raw data that was sent from the CGI script. getRawData() returns the raw HTML or any other output data from the web server as a byte array.

callOneWay() is special in that it returns immediately. This is useful for a programmer who does not care to wait for a result status. However, this does not mean that your program cannot retrieve the data that callOneWay() retrieves in a separate thread. Using the observer interface or fetchNextRecord() method discussed later in this article allow other parts of your program to get to the data that was retrieved via callOneWay().

Handling Communications Timeouts

The ability to detect when an attempt at JavaCGIBridge communications takes too long is inherent in the JavaCGIBridge class. The default timeout is 10 seconds. This can be adjusted or queried with setTimeOut() and getTimeOut() respectively. When a synchronous method which blocks until data is returned such as getRawData() or getParsedData() are called, the JavaCGIBridge object will throw the JavaCGIBridgeTimeOutException if the class takes too long to finish retrieving the data.

The only time this does not occur is with the asynchronous non-blocking method callOneWay(). Since it returns immediately before actually finishing the thread, the JavaCGIBridge object cannot use the normal Java exception mechanism to catch this timeout. For this case, there are three possible alternative actions that can be taken if an asynchronous JavaCGIBridge times out.

First, if observers are watching the class, a JavaCGIBridgeTimeOutException is passed to the observer inside of a JavaCGIBridgeNotify object if a timeout has occurred. Second, if you use the

JavaCGIBridgePool, it can clean up JavaCGIBridge objects that have timed out. Third, the JavaCGIBridgeExtension object's fetchNextRecord() method detects when a timeout has occurred.

Data Parsing Separators

To support parsing more effectively, the JavaCGIBridge core class allows changing the data delimiters. The default start of data delimiter is "<!--start of data-->\n". The default end of data delimiter is "<!—end of data-->\n". This allows the parsed data to be embedded between two HTML comments in case the CGI Scripts being called are also doubling as a plain HTML forms interface to data. The default field delimiter is a pipe symbol (|) and the default record delimiter is a newline (\n).

It is important to note that JavaCGIBridge supports arbitrary delimiter sizes. Instead of just a single pipe or a single newline, you can use larger, more heterogeneous delimiters. This is useful in the case where you believe your raw data may contain a pipe or a newline. If these stray characters are a problem, a suggested field delimiter to use is something like "~|~" and a record delimiter of "<~>\n". These are less likely to occur in plain text output but are still relatively short and computationally easy to parse.

Since JavaCGIBridge supports parsed fields being returned as raw byte arrays instead of the default java Strings, setting even longer delimiters is useful for returning binary data such as a gif or jpeg image. In this case, it is recommended to set a delimiter to something MIME-like such as "___CabbageNeck__Carrots_XandX_corn" or some other string which is unlikely to show up in your binary data.

Asynchronous Calls

While most programmers will simply be interested in retrieving small or medium size result sets which do not take much processing time to retrieve, there are occasions where having the application wait and do nothing while retrieving the data is not a user-friendly option.

First, if the result set is large such as one hundred thousand records, this may entail a lot of network time transferring data to the applet. In addition, if the user is getting a large query result back, they may discover after getting all the data that it was not the information they really wanted. In this case, it is useful to support a mechanism for canceling the communication. The same issues may hold true for small result sets if the speed of the internet connection of the client may have a possibility of being slow.

Another likely scenario is one in which the applet may want to send some user requested change back to a CGI script but does not necessarily care if the change happens to get lost on the network. In this case, it is wasteful to wait for a return result if the program does not care about one.

In these cases, instead of waiting for getParsedData() to return the entire set of parsed records or getRawData() to return the entire raw HTML in a byte array, you can use the callOneWay() method to launch the CGI application and return immediately.

Of course, returning right away is useful in the scenario where you do not care about result sets, but if you do want to still know the final outcome while returning right away, there are other options. One option is that you can use JavaCGIBridgeExtension's fetchNextRecord() method to iterate through the returned result set. This option is discussed later under the design notes of the JavaCGIBridgeExtension class.

The other option is that you can set up your applet to "observe" the JavaCGIBridge as results arrive using the java.util.Observer interface. This is useful because it allows many different observers to attach themselves to a particular JavaCGIBridge. For example, you could have a graph update itself periodically while displaying the raw data in a list box in another area of the applet without having one user interface element explicitly control the other.

With Java's built-in observer interface adding additional observers is easy. Merely implement the observer interface and then call JavaCGIBridge's addObserver() method.

Observer Interface

JavaCGIBridge inherits from Observable. This allows Observers to add themselves to the class if they wish to be notified of events using the addObserver() method. Observers can remove themselves when they are done by called the deleteObserver() method.

Observers that add themselves to the object can be notified of three events. Of these three events, two are optional—being notified when a certain number of records have been parsed and being notified when a certain number of bytes have been received. The remaining event is basically a time out event that is sent to all observers when the JavaCGIBridge times out.

Of the user-registered events, the first is a method that allow the programmer to specify how many records have to be parsed before sending a notification—setParsedNotifyInterval(). For example, if this value is set to ten records, then a notification will be sent to observers every time ten more records are parsed. The message that is sent is actually a JavaCGIBridgeNotify object that contains the entire Vector of Vectors parsed so far. The interface to the JavaCGIBridgeNotify object is shown in Table 3.

Table 3 - JavaCGIBridgeNotify Public Methods

Category	Method Name	Function
Data Retrieval	getData	Generic function which returns whatever data was sent inside it. This will be a Vector for parsed data or a byte array for raw data. If the JavaCGIBridge timed out, then this will return a JavaCGIBridgeTimeOutException object.
Data Retrieval	getRawData	Returns the passed data as a byte array. If the data was not sent as a byte array, null is returned.
Data Retrieval	getParsedData	Returns the passed data as a Vector. If the data was not sent as parsed data, null is returned.
Boolean Flag	isAtEndOfData	Returns true if this is the last observed event for the JavaCGIBridge (end of data reached).
Boolean Flag	isTimedOut	Returns true if the JavaCGIBridge timed out before the observed event occurred

The user of a JavaCGIBridge object can also specify how many raw HTML bytes are received before sending a notification using setRawNotifyInterval(). Just like the parsed records notification, observers are sent a JavaCGIBridgeNotify object. However, instead of Vector data, the notify object contains byte array data of the raw content that has been received so far.

Both the parsed and the raw HTML notification also send an additional update to observers at the very end of doing the parsing or reading the raw HTML content. This is also true as long as the value for the interval to be notified is not zero.

This convention has the added bonus of allowing a programmer to get notified at the end of the entire data retrieval in place of regular intervals. Setting the interval to a negative number will make sure that updates will not be set at regular intervals. However, since a negative number is not zero, a notification will be sent when the whole data has been parsed or retrieve as raw content in a byte array.

The decision to overload the interval with this “negative value trick” was done in order to save space in the core library. Otherwise, to accommodate this flexibility would have meant adding separate “end of data notification” flag variable, code for checking for this flag, as well as separate accessor and mutator methods to view and change the flag.

The release of the version 1.1 java compiler added support for inner classes in Java source code. If you are using a JDK 1.1 or higher Java compiler, you may want to consider using inner classes which implement the observer interface as a means to allow a single applet to observe the behavior of more than one JavaCGIBridge object. While it is possible to keep track of the objects manually within a single update() method, inner classes can result in cleaner code that avoids all sorts of if-statements checking for object

types. This is very similar to the idiom of using inner classes for adding event listeners in the JDK 1.1 AWT event model.

Observers are sent a time out update whenever the JavaCGIBridge communications times out. The observer is still sent a JavaCGIBridgeNotify object except that it wraps around a JavaCGIBridgeTimeOutException object instead of real data. To detect this, the update() event should be coded to always check the JavaCGIBridgeNotify object using the isTimedOut() method.

Accommodating Asynchronous Communications Design Patterns

The use of an observer interface to only catch when the end of data retrieval is reached is basically an adaptation of the Distributed Callback pattern described in Thomas J. Mowbray and Raphael C. Malveau's CORBA Design Patterns book. Although this is not CORBA, JavaCGIBridge is treating the web server as middleware so many general distributed programming design patterns still apply.

Another pattern from CORBA Design Patterns that can be adapted for use with JavaCGIBridge is the Partial Processing pattern. Partial Processing basically mitigates problems with long network latency or server processing time by allowing partial results to be returned in one process so that a call to a second process which relies on the completion of the former can be accomplished more quickly.

To implement this type of pattern in JavaCGIBridge, you could launch one JavaCGIBridge via the callOneWay() operation and set up an Observer interface which will catch the first results that come in and do something with them. Meanwhile, before the first JavaCGIBridge has completed but after getting some relevant results which tell the applet what to do next, a second JavaCGIBridge can be launched which asks for the second part of the data that relied on the first pieces of data being sent over the network.

Since partial processing is difficult to visualize, it is useful to work through an example. One example might apply in the realm of distributed image processing. Let's assume that you have a web server which contains the processing power and logic to perform complex image processing. Furthermore, assume that the general length of a communication to and from the web server is going to be relatively slow—one second over the Internet.

In this example, the applet needs to perform some processing on an image. However, it needs some information about the image before it does that processing. In addition, this processing is not predetermined. Perhaps the order or applicability of applied image filters changes depending on some heuristic such as how degraded the image is.

Furthermore, in this example, there is preprocessing that can be performed on an image before the actual application of filters that is the same for all images but since the images are dynamic, the preprocessing cannot be cached ahead of time.

The time it takes for the server to get the image is 2 seconds, the time it takes to do preprocessing takes 10 seconds, and the time it takes to apply the chosen filters is 10 more seconds on the server. In this case, if you did each operation in sequence using synchronous calls, you would end up taking 28 seconds.

The first step takes 1 second to make the first CGI call, 2 seconds to produce the image, and 1 second to return the image back to your applet. The second step takes 1 second to make the second CGI call for preprocessing, 10 seconds to perform preprocessing, and 1 second to let the applet know preprocessing was done. The third step takes 1 second to call the third CGI process, 10 seconds to do the filtering, and 1 second to return the results. Thus, the total processing time is 28 seconds.

Of course, if you were stuck with synchronous calls, you would probably just combine the preprocessing step with the filtering step. This reduces the processing to just two CGI calls and results in 26 seconds. However, using the Partial Processing pattern can still help shave some time off this result.

The obtaining of the image and the preprocessing step would be combined in one CGI program. Then, callOneWay would call this CGI program and be set up to use the observer pattern to get the initial image data as soon as it is ready. When the observer update() method sees that it has the image, it immediately launches a second asynchronous JavaCGIBridge request to obtain the results from the filtering step.

Of course, the second request will not return any data until the first program has completed the preparsing step, but this is not important. What is important is that several seconds of network lag have been shaved off because the final request was made ahead of the preprocessing being completed and because the preprocessing did not have to be requested through a separate CGI request.

At the time the second request is made, four seconds have passed. The last second was due to the network lag in transferring the image back to the applet. However, in that last second, the preprocessing step had already continued on! Thus, only nine seconds are left for preprocessing at this point. Then, at the four second mark, the filtering CGI request is made. This takes another second to reach the server, so by the time it reaches the server, the preprocessing step has only 8 seconds left. Thus, the second CGI request will wait for 8 seconds for preprocessing to stop and then go for 10 seconds and then return the results for another second.

With partial processing, the total round-trip time is now four seconds for the initial request with only twenty seconds for the second request. No time is wasted in the network trip for the second request being launched because preprocessing was continuing while this was happening. Thus, the total time is 24 seconds. Note that as the network lag becomes worse, the use of the partial processing pattern will help even more. Figure 4 shows this image-processing example.

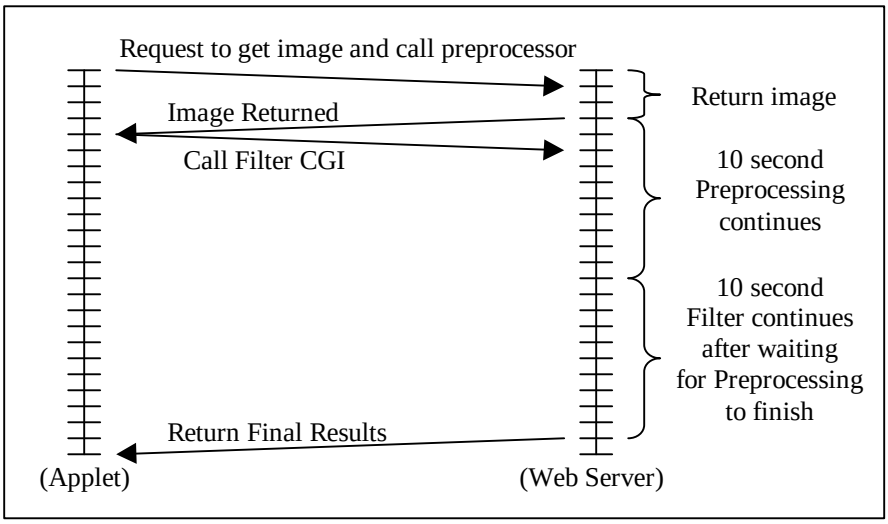

Figure 4 - Partial Processing Example

It is worth noting that the same basic pattern can be achieved by calling getParsedData() instead of using the Observer interface if the Web Server can accommodate a flexible multi-threading or multi-processing model. Instead of using the Observer mechanism to get the first partial result set, you could code the CGI script to launch another process or thread that continues processing that the second request will be dependant on. Then, the original CGI process can return the first partial result immediately. Since the CGI program has officially "returned" at this point, getParsedData() will return the first result set and the second request can be made.

Reading Setup Files

The core JavaCGIBridge class supports decoding of rudimentary setup files written in HTML. Basically the JavaCGIBridge decodes the HTML file the same way it would decode the output of a CGI script. A helper method, getKeyValuePairs(), then takes the resulting Vector of Vectors and creates a Hashtable whose keys and values correspond to the parameters and parameter settings respectively.

While an accepted mechanism for setting up an applet consists of coding PARAM tags for the applet, this method tends not to be very flexible. For one thing, if you call the applet from multiple HTML pages, you have to duplicate the PARAM statements over and over again.

Second, using PARAM tags makes it hard to do testing of parameters. Separate “development”, “test”, and “production” HTML files that contain the applet all need to have their PARAMs copied. Unfortunately, the change to the HTML display around the applet will result in having to coordinate HTML display changes with the development, test, and production HTML files.

Separating the PARAMs into external HTML files whose sole purpose is to provide setup information to the applet resolves this problem. With this method, the HTML display around the applet is decoupled from the actual setup behavior of the applet. Thus, it is easier to have separate HTML and Java programmer development environments.

Retrieving Binary Data

Calling getParsedData() normally returns a Vector of Vectors where each field in the Vector is a Java String object. However, it is recommended that you change the encoding to a byte array using the setParseAsRaw() method if you wish to retrieve binary data.

Remember that if you wish to retrieve binary data, the likelihood is high that a single character data delimiter might be matched inside the binary data simply by chance. Therefore, it is recommended that you increase the delimiter sizes before parsing binary data as it is suggested in the previous section discussing the changing of data separators.

JavaCGIBridgeExtension Design

The JavaCGIBridgeExtension class was designed to include extra features that did not belong in the core class. These included methods that were more specialized and so did not need to be in every JavaCGIBridge program. For example, object serialization is limited to JDK 1.1 browsers, so I did not want to penalize users of the class that were not likely to use object serialization if they were using server-side languages other than Java.

JavaCGIBridgeExtension basically inherits from JavaCGIBridge so it contains all the same interfaces. These features include serialized object support, rudimentary result set iteration support, and a showDocument() method that improves upon the method built into the Java JDK’s AppletContext class. Table 4 contains a list of the methods JavaCGIBridgeExtension adds to the framework.

Table 4 – JavaCGIBridgeExtension Public Methods

Category	Method Name	Function
Data Output	getObjectOutputStream	Returns an ObjectOutputStream that can be used to output serialized objects to a CGI application
Data Output	getByteArrayOutputStream	Returns a ByteArrayOutputStream that can be used to output raw binary data to a CGI application
Data Retrieval	getObjectData	Calls a CGI Application and returns an ObjectInputStream from which serialized objects can be retrieved
Data Retrieval	fetchNextRecord	Fetches each records as a Vector of fields. When it returns null, there are no more records to retrieve.
Helper	showDocument	Commands the browser to open a new document. Fixes a Mac Netscape bug and also provides the capability of sending CGI parameters to a script using the GET method automatically.

Raw Data Output

Normally, to output data to a CGI script, you call a function such as callOneWay(), getParsedData(), or getRawData() and pass it a Hashtable containing form variables. However, this only works well for regular strings. To pass binary/raw data to a CGI script, JavaCGIBridgeExtension gives you the capability of creating the byte array that will be sent to the CGI script directly.

Calling getByteArrayOutputStream() returns an output stream that you can write() bytes to for sending over the HTTP protocol. After this writing has been done, simply calling callOneWay(), getParsedRecords(), or getRawData() methods without a passed form variable Hashtable will result in the JavaCGIBridge using the ByteArrayOutputStream to send data out instead of the normal form variable/value pairs.

Serialized Object Support

For programmers that do not mind restricting their users to JDK 1.1 compliant browsers, sending serialized objects to CGI programs written in Java or to Java Servlets can be a useful way of implementing a form of marshalling strongly typed Java data for "free".

Traditional POST data is simply sent as strings. A programmer typically has to do a lot of manual coding to test in a CGI script whether the string that was sent is an integer or a float or even a date type of object. However, by passing serialized objects, this data type checking is done for you.

Of course, it is not my intent to say that strongly typed data marshalling is better than weakly typed marshalling—traditional POSTing of strings. Weakly typed marshalling tends to have advantages such as being more flexible and resilient to small changes in protocol and such properties as field sizes.

On the other hand, without strong runtime checking, changes in one side of the code can result in subtle errors. For example, a date object (12/10/98) could be seen as three divisions if a number is expected in that field within a CGI script.

There are two method calls to support the sending and receiving of serialized objects. To send serialized data to a URL, use getObjectOutputStream() to obtain an ObjectOutputStream to write objects to. Just like getByteArrayOutputStream discussed previously, writing to ObjectOutputStream will cause this data to be sent to the URL when you finally execute callOneWay(), getParsedData(), or getRawData().

To get serialized objects from the URL, you can do one of two things. One way is to use getRawData() as the method to get the URL data. This will return a byte array which you can then turn into a ByteInputStream. This, in turn, can be turned into an ObjectOutputStream using the traditional Java I/O decorator pattern.

However, you may find it easier to use JavaCGIBridgeExtension's getObjectData() method. This method follows the same parameter format as getRawData() except it creates the ObjectOutputStream for you. The following is some sample code showing how to send and receive objects from a JavaCGIBridgeExtension object.

```
try {
    ObjectOutputStream oos = jcbe.getObjectOutputStream();
    oos.writeObject(so);
} catch (IOException e) {
    System.out.println("IOException adding Object");
}
ois = jcbe.getObjectData(u);
ExampleObject eo = (ExampleObject)ois.readObject();
```

Parsed Results Iterator

The fetchNextRecord() method allows iteration through result sets without using the observer interface. To use this, initiate a retrieval of result sets using callOneWay() and then within a while loop, call fetchNextRecord() until it returns no more records (null). An example of code that does this iteration appears below. The elements of Vector v contain all the fields of the record.

```
Vector v;
```

```
try {
    while (null != (v = _threadJCBE.fetchNextRecord())) {
        count++;
        _progressPanel.setCurrentRecord(count);
    }
} catch (JavaCGIBridgeTimeOutException e) {
    System.out.println("JavaCGIBridgeTimeOutException Thrown!!");
}
```

In using this iteration method, you should weigh the fact that there is a disadvantage to this iteration approach when using the Java abstract windowing toolkit (AWT). Most JavaCGIBridge interactions will likely be triggered from within an AWT event such as a button click or a selection.

However, if you spend a lot of time in an event iterating through the records, the AWT will be unresponsive to other actions such as repainting a progress bar. In this case, it is necessary to launch a thread that iterates through fetchNextRecord() and exits the event handler quickly. If you are going through the trouble of creating a new thread, it is advisable to simply go ahead and use the observer interface discussed in the core JavaCGIBridge class design section instead.

Show Document

One of the major disadvantages of Java applets is the lack of a really robust printing mechanism. Yet nearly any major Intranet application will require some sort of report to be printed. Using applets in a browser can solve this problem by simply telling the browser to open a new browser window with the report laid out as standard HTML.

The AppletContext class inside of the java.applet.* hierarchy allows you to do this. However, JavaCGIBridgeExtension expands this method to provide several advantages. The first is a bug fix for the way Mac Netscape v3.x behaves with showDocument(). The second deals with the passing of form values to a CGI script via showDocument().

The bug with showDocument() is that on some browsers, instead of opening a blank browser window and displaying the passed URL, you just get stuck with a blank browser window. It turns out that the way to fix this bug is relatively simple. All you need to do is call AppletContext's showDocument() twice in order to trick the browser into filling in the blank window.

This fix works for a unique target such as "_myTarget" since only one browser window can have that unique target. Thus, the first time "_myTarget" is called, a blank browser window is opened. The second time "_myTarget" is called, the blank browser window is filled in with the URL.

Unfortunately this fix does not work for targets of "_blank" which always opens a new blank browser window. On browsers that have this bug, calling the "_blank" target twice simply results in a second blank browser window instead of filling in the first one through the second showDocument() call.

To fix the "_blank" target bug, JavaCGIBridgeExtension uses a trick. Basically passed "_blank" targets are translated to a random target name plus an appended count of how many times showDocument() was called. The amount of times showDocument() was called in JavaCGIBridgeExtension is tracked using a static class variable.

This random value plus the count results in a "unique" target that simulates opening a plain "_blank" target. Since the "_blank" target is really a uniquely generated target name, calling showDocument() twice trick works. A further reference on this bug may be found at http://gunther.web66.com/crossjava/.

The second advantage of JavaCGIBridgeExtension's showDocument method over the one in AppletContext, is that it accepts the JavaCGIBridge standard way of passing form variables using a Hashtable. This form variable Hashtable is translated into a URL encoded string which is appended to the CGI script URL. Thus, JavaCGIBridgeExtension's showDocument is more geared towards opening CGI scripts and passing parameters that result in more sophisticated reporting capabilities for an applet interacting with CGI scripts.

JavaCGIBridgePool Design

JavaCGIBridgePool is ultimately a manager class. Its sole purpose is to manage large numbers of JavaCGIBridge and JavaCGIBridgeExtension objects.

If you are programming a small applet with only a few CGI calls, using JavaCGIBridgePool is probably overkill. However, if you believe you will be accessing CGI scripts during the majority of your application, JavaCGIBridgePool will manage several of the more complicated aspects of a large JavaCGIBridge applet. Table 5 contains a list of the methods used in the JavaCGIBridgePool.

Table 5 - JavaCGIBridgePool Public Methods

Category	Method Name	Function
Obtain Object	getJavaCGIBridge	Returns a free JavaCGIBridge object
Obtain Object	getJavaCGIBridgeExtension	Returns a free JavaCGIBridgeExtension object
Utility	cleanPool	Removes objects in the pool that have timed out.
Utility	clearPool	Removes all objects in the pool that are free or have timed out.
Utility	removeJavaCGIBridge	Removes the passed JavaCGIBridge reference from the Pool
Property	get/setMaxObjects	Determines the maximum objects that can exist in the pool. The default is 0 (no max). If a user requests an object and all of them are busy and this limit is reached, the application is blocked until an object gets freed.
Property	get/setMaxSpareObjects	Determines the maximum amount of IDLE objects to keep around in the Pool. The default is 0 (no limit). When cleanPool is called, if there are more idle objects lying around than this value, they will be removed from the object pool.
Property	get/setCleanTime	Determines the amount of time in milliseconds before a Thread will automatically launch the cleanPool method. The default is 0 (no thread is created to clean up idle or timed out objects).
Default Property	get/setDefaultStartDataSeparator	Get and set the default start of data separator for parsing
Default Property	get/setDefaultEndDataSeparator	Get and set the default end of data separator for parsing
Default Property	get/setDefaultFieldSeparator	Get and set the default field separator for parsing
Default Property	get/setDefaultRecordSeparator	Get and set the default record separator for parsing
Default Property	get/setDefaultParseAsRaw	Get and set the default flag for whether to parse fields as Strings or byte arrays in JavaCGIBridge
Default Property	get/setDefaultTimeOut	Get and set the default timeouts for JavaCGIBridge objects
Default Property	get/setDefaultRawNotifyInterval	Get and set the default raw notify interval for JavaCGIBridge objects
Default Property	get/setDefaultParsedNotifyInterval	Get and set the default parsed notify interval for JavaCGIBridgeObjects

Reasons For Using JavaCGIBridgePool To Manage Objects

First, if your design calls for using the asynchronous callOneWay() method frequently in the applet, you cannot reuse the JavaCGIBridge object until the getStatus method says it has stopped processing. If you plan on reusing JavaCGIBridge objects, the JavaCGIBridgePool will check automatically to see if there is an idle object. If there is an idle object, it gets handed to your program. If there is no idle object, a new JavaCGIBridge is created, added to the pool, and handed to your program.

Additionally, even if your design calls for not reusing JavaCGIBridge objects, your applet may still run into problems. If your asynchronous calls actually time out, there is a likelihood that no one will be notified of the problem. The JavaCGIBridgePool supports the creation of a background thread which periodically calls the cleanPool() method to clean out any objects that have timed out and notify observers of that event.

Finally, there is a host of default property methods that allow JavaCGIBridgePool to automatically setup a JavaCGIBridge object differently from the built-in defaults. For example, the default field delimiter is a pipe symbol (|). You can set the default field delimiter to something else in JavaCGIBridgePool such as a comma. Then, all the created JavaCGIBridge objects will be created with a comma-based field delimiter instead of the built-in default.

JavaCGIBridgePool Behavior Properties

In order to accommodate a variety of object reuse models, there are several other properties that can be set. If your applet occasionally has spikes in the amount of asynchronous calls it wants to make, you can limit the amount of objects that get created using setMaxObjects(). The default is 0 for unlimited objects. If all the objects are busy and the max object limit is reached, the JavaCGIBridgePool will force the program to wait until one is freed up.

This may seem like an unusual property but it can be a useful precaution. Some platforms view sockets as a limited resource. An out of control applet could easily hog all the network resources of the client computer if this property is left unset.

A less stringent version of max objects can also be set. setMaxSpareObjects() tells the applet how many idle objects to keep hanging around. When cleanPool() is called manually or through the cleanup thread, any idle objects above max spare objects are destroyed. Like setting the maximum number of objects, the maximum spare objects property is also useful in case of spikes in JavaCGIBridge object usage.

However, rather than protecting resources like setMaxObjects(), setMaxSpareObjects() protects you from stealing RAM away from the applet by having so many objects lying around for long periods of time. By default MaxSpareObjects is set to 0 which means unlimited spare objects are allowed.

The final management property that can be set is the number of milliseconds before a clean up thread wakes up and cleans out timed out or idle objects as indicated by setMaxSpareObjects(). setCleanTime() is by default 0. This means that there is no additional thread cleaning up objects. Setting this value to a positive number launches the clean up thread and programs it to wake up at the passed interval in order to call the cleanPool() method.

Singleton Design Pattern

JavaCGIBridgePool is implemented following the Singleton pattern. However, what may appear unusual is that unlike many Singletons such as Java's System class, it is not implemented using static methods and class variables. JavaCGIBridgePool is designed to be instantiated.

There are several reasons for this. The most important reason is that it is possible that several different applets on a page may all want to use JavaCGIBridgePool. If JavaCGIBridgePool were implemented as a class-based singleton, then all the applets would be forced to use the single JavaCGIBridgePool. This would cause all sorts of subtle problems if the setDefaultXXXXX() property methods expected different behaviors in different applets.

The second reason is that in a large enough applet, there may be a design reason to have more than one pool. For example, if you have an applet that makes heavy use of asynchronous calls in one section of the

applet and only a few asynchronous calls in another portion of the applet, it may make sense to create two pools. One pool would be set up to guard asynchronous calls very closely and limit the amount of objects created so that the excessive communications part of the applet is mediated. While this is happening, the other pool would be set up to be more liberal in its constraints and allow the long-term pattern of application usage to govern how many objects to keep around.

While JavaCGIBridgePool does support these sophisticated scenarios, most applications will likely use JavaCGIBridgePool as more of a helper class for setting defaults and helping manage object usage. Unless you have specific design decisions, the typical usage of this class would involve instantiating just a single instance for each applet.

Exception Handling Design

The JavaCGIBridge framework handles two exceptions. One of the exceptions is thrown if the JavaCGIBridge communications time out. The other is thrown if the JavaCGIBridge object is already busy handling another request.

JavaCGIBridgeTimeOutException

The time out exception is called JavaCGIBridgeTimeOutException. It is a checked exception. Synchronous methods such as getRawData() and getParsedData() must be encapsulated in a try/catch block. This was done because time outs are generally caused by network problems that the programmer has no control over. Thus, I wanted to make sure that programmers were forced to always take this possibility into consideration.

The JavaCGIBridgeTimeOutException object is also passed as a data element of the JavaCGIBridgeNotify object if observers are busy observing a JavaCGIBridge object that throws this exception. The reason for this is that since callOneWay() is an asynchronous method, there is no way to encapsulate it in a try/catch block to catch an exception.

Exception handling is tricky when dealing with multi-threaded code. However, using the JavaCGIBridgeNotify mechanism of notifying observers of a problem seemed better than the alternative. In the alternative, I could have made a new ThreadGroup object that handled uncaught exceptions in the communications thread and created a callback mechanism to allow custom exception handlers to be referenced in the ThreadGroup object. However, if I used this design it would have meant adding a lot more code to the core part of the framework and also would have made the use of the class library more complicated for average use.

JavaCGIBridgeStillProcessingException

The JavaCGIBridgeStillProcessing exception is more deterministic, so it is implemented as an unchecked exception. You do not have to use a try/catch block around methods that throw this exception. As a programmer, it is your responsibility to check the getStatus() method to see if the object is still in use before reusing it. You can also use JavaCGIBridgePool which does this checking automatically.

In simple applications where new JavaCGIBridge objects may be created and destroyed for each request, this exception will not have an occasion to be thrown. Generally, it might be thrown if you are reusing the JavaCGIBridge objects within a multi-threaded application without checking the status of the object or using a manager such as JavaCGIBridgePool.

Having a JavaCGIBridge object throw the still processing exception is an indication of programmer error that can be avoided. This is in opposition to an exception that occurs because of the external environment such as the time out. Since there are many ways for the programmer to avoid this exception, I did not want to force programmers to catch it.

Examples of JavaCGIBridge Framework Usage

There are eleven examples that show the various ways the JavaCGIBridge framework can be used. They can be located along with the full source code and JavaDoc of the framework at http://gunther.web66.com/JavaCGIBridge/.

All the examples except for the object serialization one use Perl scripts for the CGI portion. I used Perl because it is the defacto standard CGI scripting language in use on the Web. This makes it much easier for users to download the JavaCGIBridge framework and try it out on their own servers or on their Internet Service Providers which may not support server-side Java but almost certainly support server-side Perl.

The first two examples are simple front-ends to some Perl scripts that perform a calculation and return a result. The first example takes as string, sends it to a CGI script which reverses the string and then sends it back. The applet then displays the reversed string.

The second example displays a bar chart of a list of numbers. When the "call CGI" button is clicked, the values are sent to a script which raises all the numbers to the power of two. These numbers get passed back to the applet and are subsequently graphed.

The third example demonstrates simple parsing of records by printing out the parsed results in a text area. The fourth example shows the retrieval of a simple raw HTML document output from a CGI script.

The fifth example uses JavaCGIBridgeExtension. It shows how JavaCGIBridgeExtension deals with object serialization by sending an object to a CGI application written in Java. This CGI application sends back several other serialized objects back to the applet. The results are displayed in a text area on the applet.

The sixth example builds on the fifth one by generically using JavaCGIBridgeExtension to send and receive raw image data rather than serialized objects. One gif image and one jpeg image are transferred to the applet from a CGI script and then sent to another CGI script from the applet.

The seventh example shows an example of using the Observer interface to implement a progress bar. The eighth example is the same applet but using the fetchNextRecord() iterator to accomplish the same task. The ninth example uses the Observer interface to keep a watch on the raw data (HTML) that has been received so far.

Example ten demonstrates the use of JavaCGIBridgePool to manage objects. It uses setMaxObjects to show how setting it too low can affect application performance if the CGI scripts that are being called are too slow. In this example, the CGI scripts use the sleep() command to simulate long running scripts with one to six second delays.

Finally, example eleven shows the use of the showDocument() helper method in the JavaCGIBridgeExtension class.

Discussion

The JavaCGIBridge framework contains a simple API for getting parsed data from a CGI script or Java Servlet. Additionally, it also contains a rich set of features to accommodate a variety of distributed programming patterns.

The use of asynchronous calls and the Observer design pattern allow adaptations of Mowbray and Malveau's Distributed Callback and Partial Processing patterns to be implemented. A full-featured object pool accommodates JavaCGIBridge object reuse models and is especially useful for managing JavaCGIBridge objects in a multi-threaded environment or a program where you are using a lot of asynchronous calls to callOneWay().

All of this is accomplished by simply using Java's core URLConnection class, so applets have full access to SSL and other browser-integrated HTTP communications features. The core class library also succeeds in being very small compared to JDBC, CORBA, or other middleware libraries for Java.

Even the full JavaCGIBridge framework combined is much smaller than these libraries. The full JavaCGIBridge framework actually contains a lot of communications features for "free" that you would have to program yourself if you were using another communications framework such as JDBC. Observing

huge results sets as they come in asynchronously and managing pools of JDBC connections are generally left as an exercise for the diligent distributed Java programmer using the JDBC API.

There are cases in server-side programming where using JDBC or middleware such as the Java EnterpriseBeans standard makes a great deal of sense. Unfortunately, applets pose more constraints in that the classes downloaded for the applet should remain as thin as possible while still providing security capabilities such as SSL. In addition, it is useful to be able to leverage previously programmed business logic from legacy CGI code. The JavaCGIBridge framework was designed to accommodate these constraints and provide a useful alternative to traditional distributed Java programming libraries.