Department of Computer Science, UTSA
Technical Report: CS-TR-2008-015

GPLBrowse: Architectural Overview

by
Cory Burkhardt
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 72849
cburk@cs.utsa.edu

Abstract

GPLBrowse is a complex web application that provides an online environment for exploration of bioinformatics data. This data, which was gathered from the Gene Expression Omnibus or NCBI GEO (http://www.ncbi.nlm.nih.gov/projects/geo/) project, is presented in GPLBrowse as a series of scatter plot charts. On the server side, GPLBrowse is hosted by a Tomcat web server running a Java servlet. On the client side, the web browser executes JavaScript code that was built using Ajax technologies with the aid of the Yahoo User Interface library (YUI). This technical report describes the server side and client side designs of GPLBrowse, and the interactions between the two.

1. Introduction

GPLBrowse is a complex web application that provides an online environment for exploration of bioinformatics data gathered from the Gene Expression Omnibus or NCBI GEO (http://www.ncbi.nlm.nih.gov/projects/geo/) project. The application relies on a split client/server architecture to present a rich user-interface that is compatible with most modern web browsers. GPLBrowse provides the user with the capability to explore collections of charts, instantly examine metadata associated with the plotted data, and generate and download customized data from a chart.

On the server-side, GPLBrowse runs in the context of a Java servlet operating within an Apache Tomcat web server. The information flow in GPLBrowse is illustrated in Figure 1. Some offline preprocessing must be performed before the server can go live with the data. This process involves using the bioinformatics data files generated by GPLGeneMap to build a metadata database and to generate XML and data files describing the charts that will be hosted by the server. Once this offline preprocessing is complete, the Tomcat server can then be brought online to host this data along with the HTML and JavaScript web content that defines the client portion of the application.

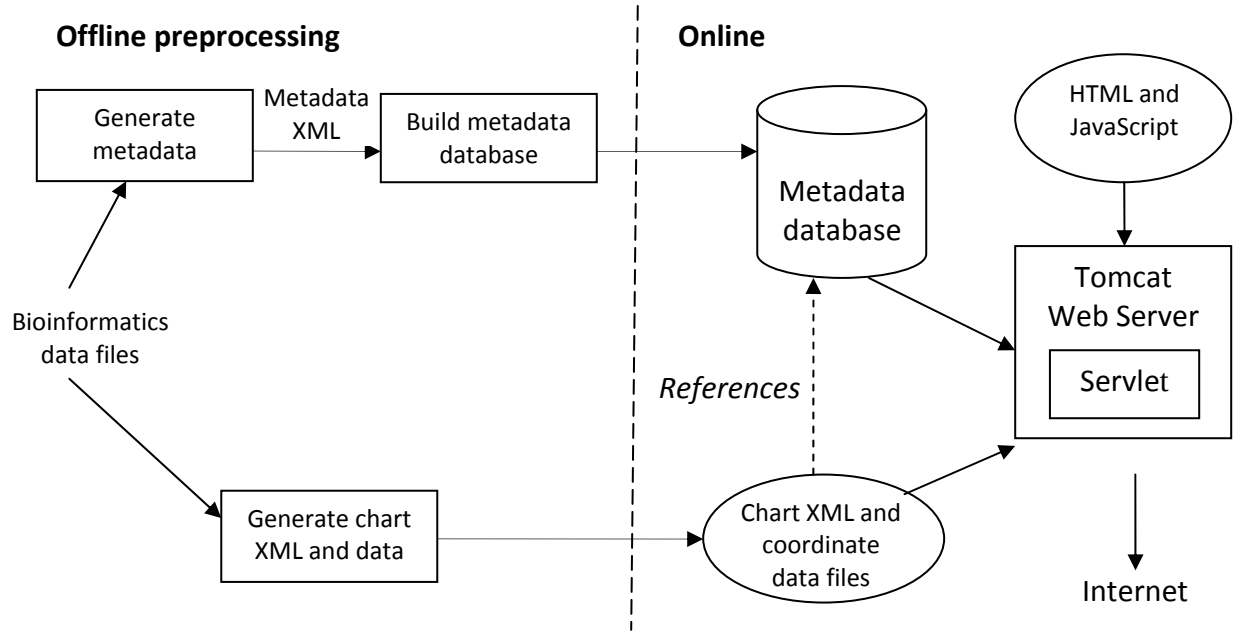

Figure 1: Offline preprocessing is performed to generate a metadata database and chart XML and data files from the bioinformatics files generated by GPLGeneMap. This content is then hosted by a Java servlet in an Apache Tomcat web server.

2. The server-side implementation

In addition to hosting the HTML and JavaScript files that are downloaded by the client browser, the server-side portion of GPLBrowse is responsible for hosting dynamic content that can be accessed from the browser. This content, which consists of text documents, XML, images, and downloadable data files, must be constructed based on the request query parameters and post data sent from the client browser. The details of this process and the responsibilities of the server are discussed below.

2.1 Java Servlets and the Apache Tomcat web server

The server-side portions of GPLBrowse that produces the dynamic content is implemented using Java servlet technology. Java servlets are a relatively low-level technology developed for the Java programming language that give programs total control over the response that is sent back to the client in response to a request. When a request to access a servlet is received by the web server, it loads the servlet if it is not already loaded, and then forwards the request to the servlet after a minimal amount of processing. The servlet is then free to send a response back to the client with any kind of content (e.g., plain text, XML, or images).

The dynamic content served in GPLBrowse is processed by dozens of request handlers that are all accessed through a single servlet. One parameter, called the action parameter, is required with all requests to the servlet. This parameter tells the servlet which request handler should process the request. For instance, an action parameter of "`chartxml`" tells the servlet that the request handler that serves chart XML should handle the request.

Another parameter, called "`sid`" for session ID, may also be supplied with any request to ensure that the server has not restarted since the time that the client program was loaded into the browser. This is done because the chart XML or other types of content may change when after the server restarts. When the server first loads, it generates a string of text based on the current system time. The client JavaScript program retrieves this value when it first loads and then passes this value as the "`sid`" parameter to the server with most of its requests. The servlet compares the supplied value with the server's current session ID. If the values match, the request is forwarded to the appropriate request handler. If the values do not match, an error response is sent back to the client.

The servlet must be hosted in a web server that supports Java servlets. The Apache Software Foundation has developed such a server, which it calls Tomcat. This is the web server that hosts GPLBrowse. Tomcat is open source and can be run on Windows, Linux, and other platforms. It is configured to load the GPLBrowse servlet through XML configuration files. A home directory location on the file system is also configured in the XML. When requests for static content (e.g., HTML, CSS, JavaScript, or image files) are received, the content is loaded and served from this home directory. These requests are handled directly by the web server and are not passed to the servlet.

2.2 Charts and data

The GPLBrowse servlet is responsible for providing the client with access to the chart descriptor XML and the entity data that describes the points plotted in the charts. The chart XML comes directly out of the server's configuration XML. The datasets that house the coordinates for the points on the charts are stored in comma-separated-values (CSV) text files. The dataset data that is sent back to the client is also sent in CSV format, but the format of the data is reorganized based on the query parameters in the request.

All of this data is loaded from text files saved on disk. Reading this data into the server for processing is a very slow process. Furthermore, this process does not provide efficient access to small portions of the data because the entire file must be read in order to find where all of the data is located in the file. For this reason, when data is loaded by the servlet from these CSV files, it is placed into a memory/disk cache for quicker access in future requests. The Apache JCS cache library was used for this purpose. This cache is created with a maximum memory cache size (which is configurable in an XML file on the server) and employs a least-recently used strategy for its memory cache eviction policy. When an item in the cache is evicted, it is serialized to disk in a binary format so that it can be deserialized quickly back into memory the next time it is needed.

2.3 Entities, metadata, and searching

In GPLBrowse, every point that is plotted in a chart is associated with some kind of entity, such as an NCBI GEO series or a sample. Each of these items has metadata that is associated with it. Quick random access to this metadata is required because the client program can request to get the metadata for any of these items at any time. Therefore, this data is preprocessed before the server is brought online and put into an indexed database. The server can then load this data and handle requests to retrieve metadata for the chart entities in a quick and efficient manner. GPLBrowse uses the Berkeley Database Java Edition to house the metadata. This database library, which is now being maintained by the Oracle Corporation, is quicker than other SQL-based database libraries because all database queries are made using direct function calls rather than indirectly through a query language that must be parsed and interpreted before it can be executed.

Another responsibility of the server is to provide the client program with the ability to search the metadata to find entities that match user-supplied search queries. The Apache Lucene library is used to implement this feature. This library allows Java programs to create "documents" that contain one or more fields and index the terms in the fields into a search database that can be queried through a structured search language. The library allows the program to dictate whether the fields in the documents are stored so that the value can be retrieved after a search and whether the field is indexed to match the keywords in searches. The search index used by GPLBrowse is built offline so that it can be loaded and used immediately when the server is started. Each document that is added to the index refers to a single entity that can be associated with a chart point. The entity name (e.g., GSExxxx for series or GSMxxxx for samples) is added to a field that is both stored and indexed. All of the metadata for the entity is put into a separate field that is indexed, but not stored. When a client request to perform a search is received, the Lucene library executes the search and produces a list of matching documents. The entity names can then be extracted from the documents and forwarded back to the client in the response.

2.4 Dynamic images

One limitation of web-based JavaScript programs is that they are unable to directly render raster images that can be displayed on the web page. Instead, this task must be delegated to the server through the use of the HTML `img` tag. The `src` attribute of this tag can be set to a URL that initiates a request to the server to create an image. This is how chart plots and text images are displayed in `GPLBrowse`. The servlet contains request handlers that allow clients to request that chart plots, chart overlays, or text images be created. The client can supply query parameters in the URL to specify what kind of image should be generated. Unfortunately, post data cannot be used with `img` tags.

Requests to generate chart plot images are supplied with query parameters to indicate the size of the image, the shapes used to plot the points, the minimum and maximum x and y data coordinates, the ID of the datasets to use, and the indices of the rows or columns in the datasets that contain the x and y coordinates of the points to plot. A configurable maximum size constraint is put on the width and height of the image. In response, the request handler creates the image, plots the points, and sends the image content to the browser encoded with the Portable Network Graphics (PNG) image format.

The `GPLBrowse` servlet also handles requests to generate chart overlays that can be displayed on top of the base chart image. These overlays consist of points plotted with a certain shape and color. Each point that is plotted on the overlay must be specifically identified in the request. Because post data cannot be used with `img` tags, this data must be supplied in the query string. However, client browsers can limit the length of the query strings, which limits the number of points that can be put in the request. Therefore, multiple overlays may need to be stacked on top of one another in order to create the entire overlay. The pixel coordinates of each point to be plotted are supplied in the query string. To increase the amount of data that can be supplied in a single query string, these coordinates are encoded into a binary form that uses a minimal number of characters. The request handler then decodes these coordinates and generates the overlay image.

The servlet can also handle requests to generate text rendered as images. This is provided to help create a user interface appearance on the client that is uniform across all browsers. The client supplies font and image size information in query parameters. It also adds a query parameter containing the text to be displayed. The request handler then generates the text image.

2.5 File handling

Another limitation of JavaScript running within client web browsers is that it has no access to the user's file system. Any such access must go through the server as a file upload or download. The `GPLBrowse` servlet has request handlers that allow the client to download a text file to the user's computer or load the contents of a text file from the computer. For downloading, the file's text contents is supplied in the post data of the request. The request handler responds by simply forwarding the content back to the browser, but it adds the `Content-Disposition` header line that tells the browser to save the content to a file rather than display it on the page. To upload data so that the client program can read data, the client must make use of a file form control. The user must click the button to browse to the file on the file system and then click a submit button

to initiate the upload request to the server. The servlet simply responds by forwarding the file data back to the client in the response. The client can then read this content off of the web page.

The server also provides collections of data that can be packaged and sent back to the client as a downloadable archive file. Each collection of data is stored in its own directory. The collection is composed of a series of dataset files that contain data in a CSV format. The client can filter the data by supplying filter parameters in the post data. The query string is not used due to its size restriction. After filtering is applied, all of the data is added to a gzipped tar archive file and streamed to the response as a file download (again using the `Content-Disposition` header). The ICE Engineering Java Tar library is used to create the tar archive content (the standard Java libraries have GZip support). The tar/gzip content is all generated on the fly in memory as it is being streamed to the client; it is not written to disk on the server. See section 4.4, Exporting Data, for more information about how this data is filtered.

3. Stages in the lifecycle of a client session

A client session must go through a number of stages before a chart can be displayed to the user. These are outlined below.

3.1 JavaScript compression and download

The first stage of a client session begins when the user visits the charts.html web page. This web page contains several `<script>` links to JavaScript files. Each of these JavaScript files must be downloaded from the web server before the client program can run.

During development, the JavaScript source code in GPLBrowse is spread out among many JavaScript files. This helps to keep the source code organized and maintainable, which is essential in a complex application like GPLBrowse. However, if the source code remains separated into many files when it is run over the Internet, the browser must make a separate HTTP request to download each of them. This unnecessarily consumes bandwidth and increases the delay that is experienced before the application loads, degrading the user's experience. To address this issue, a Perl script was developed which can recursively iterate through a file system directory, read each JavaScript file, and write the combined contents of all the files to standard output. The output from this script can be redirected to a new JavaScript file that contains the combined contents of all the other source files.

In addition, the JavaScript code is full white-space formatting and human-readable comments that have no bearing on the functional execution of the code. This text is completely ignored by the browser's JavaScript engine that runs the code. Thus, considerable bandwidth savings can be achieved by removing this text from the source code before it is sent to the browser. Yahoo has developed a Java program called the YUI Compressor specifically to address this problem. This program is included with their Yahoo UI (YUI) library – an advanced JavaScript library that contains many widgets and utilities that make JavaScript application development a much more manageable task for developers. The YUI compressor is a text-based program that reads JavaScript source code through standard input, compresses the source code by removing the unnecessary comments and formatting, and outputs the resulting source code to standard output. Yahoo uses the YUI Compressor in their own YUI source code that they publish for public use.

When a release of GPLBrowse is published, all of the JavaScript code that drives the client application is run through the concatenation script, and the output is fed into the YUI compressor. The compressed output is sent to a single JavaScript file, which is then included in the main HTML file. Using this strategy, a significant bandwidth savings is achieved. The development source files in the current release of GPLBrowse are comprised of 37 JavaScript files and a use total of 440 Kilobytes of storage space. After running the files through concatenation and compression, we reduce this to a single file which uses 184 Kilobytes of storage space. In addition, by enabling gzip compression on the web server, further bandwidth savings can be realized.

3.2 JavaScript initialization

When the browser encounters a `script` HTML tag as it is loading the page, the browser downloads the referenced JavaScript file and executes it (assuming the tag references a file – if it is embedded directly with JavaScript, the code is executed immediately). Aside from the external library JavaScript source, all of the code that makes up GPLBrowse is organized into classes. Although the JavaScript that is adopted by browsers does not have a language structure to formally define classes, it does have features that allow code to be written that exhibits class-like behavior.

GPLBrowse defines a framework infrastructure which allows classes to be defined in a manner similar to other object-oriented languages, such as Java. This framework makes the process of defining and working with classes much easier. It also makes the code much easier to read. It does not look as nice as it does in languages that allow formal class definitions, but it is much nicer than manually configuring every class using the basic JavaScript language features. The following lists an example illustrating how classes are defined in GPLBrowse:

```
edu.utsa.Util.createPackage('edu.utsa.geometry');

edu.utsa.Util.createClass('Rectangle', 'edu.utsa.geometry.Shape',
{
   Rectangle: function(width, height) {
      this.p_width = width;
      this.p_height = height;
   },

   calculateArea: function() {
      return this.p_width*this.p_height;
   }
});
```

The first statement in the file should declare a package using the createPackage method. Then, one or more classes can be defined using the createClass method. The first parameter of createClass is the name of the class being created. The second parameter is the name of a super class. A null value may be passed if there is no superclass. The last parameter is the body of the class which defines all of the methods for the class. This class body is defined using the JavaScript Object Notation, or JSON. A JSON definition consists of a list of properties and

values. In our case, the properties are the method names and the values are the functions that define the methods. The method that has the same name as the name of the class is used as the constructor.

When JavaScript files are loaded by the browser that contain this kind of class definition, the calls to the createPackage and createClass methods append the class information to a list. Once the browser has loaded finished loading the document, this information is used to define these classes in a manner that is recognized by the JavaScript engine.

One of the classes defined in GPLBrowse, called ChartsMain, contains the execution entry point of the application. This class, defined in ChartsMain.js, contains a static method called "main". After all of the classes have been created, this main method is called to initialize and run the application.

3.3 Browsing the charts

After JavaScript initialization has completed, the first task performed by the application is to construct the chart browser tree to allow the user to pick a chart to display. Before it can initialize the tree, however, the application must get a list of available charts and the folders that contain them from the server. Thus, the first task performed by the program is to make an XMLHttpRequest to the web server to retrieve this information. The list of charts and folders is returned as an XML document. The interaction between the tree and the web server was designed so that the contents of folders in the tree is downloaded dynamically – that is, the folder content is downloaded only after the folder is expanded rather than downloading the entire contents of the tree in one large batch at the beginning. This is accomplished using the dynamic load feature of the Yahoo UI library's TreeView class. The following listing is an example of XML returned by the server for the top-level folders in the tree:

```
<chartSetList>
   <folder
      <id>2</id>
      <name>GPL72</name>
      <tooltip>Tooltip message for GPL72 folder.</tooltip>
   </folder>
   <folder
      <id>3</id>
      <name>GPL81</name>
      <tooltip>Tooltip message for GPL96 folder.</tooltip>
   </folder>
</chartSetList>
```

The `id` element of a `folder` element specifies a unique value that identifies the folder in subsequent requests to the server. The name of the folder is displayed in the tree and the tooltip is displayed when the mouse hovers over the folder in the tree. When a folder is expanded, another request is made to the server to get the contents of the folder. The following example demonstrates the XML that is returned to list the contents of a folder that contains charts:

```
<chartSetList>
   <chart
      <id>78</id>
```

```
    <name>4% trim mean vs STD (sample)</name>
    <tooltip>Sample 4% trimmed mean vs 4% trimmed standard
deviation</tooltip>
  </chart>
  <chart
    <id>80</id>
    <name>4% trim mean vs STD (series)</name>
    <tooltip>Series 4% trimmed mean vs 4% trimmed
standard deviationtooltip>
  </chart>
</chartSetList>
```

All of the elements within the `chart` elements serve the same purpose as they do in the `folder` elements.

3.4 Displaying a chart

After selecting a chart and clicking the display button, the client-side program begins to take steps to display the chart. The program contacts the server using XMLHttpRequests to obtain information about the chart and a listing of the chart's coordinates. The chart XML that is retrieved from the server is given verbatim from the chart definition in the server configuration XML:

```
<chartDetails id="GPL72_Sample4PercentTrimMeanVsSTD">
  <id>GPL72_sample_mean_std_exp</id>
  <title>GPL72 (fruit fly): …</title>
  <description>A long description…</description>
  <tooltip>A tooltip to display on the chart</tooltip>
  <axisGroup>
    <xAxis>
      <label>4% trimmed mean</label>
      <Min>-500.0</Min>
      <Max>3000</Max>
    </XAxis>
    <yAxis>...</yAxis>
    <plot>
      <type>Sample</type>
      <datasetRef>
        <datasetId>GPL72_Sample4PercentTrimMean...</datasetId>
        <vectorType>column</vectorType>
        <xIndex>0</xIndex><yIndex>1</yIndex>
      </datasetRef>
      <style>
        <marker>o</marker>
        <markerSize>6</markerSize>
        <markerColor>0.000, 0.000, 0.000, 0.000; 0.000,
             0.000, 0.000, 1.000</markerColor>
        <highlightColor>...</highlightColor>
        <selectionColor>...</selectionColor>
      </style>
      <legendItem>Samples (1014 points)</legendItem>
    </plot>
    <plot>...</plot>
  </axisGroup>
```

```
    <keywordOrder>
      <keywordCategory>
        <name>normalization</name>
        <order>mas,rma,gcrma,dchip,vsn,unknown</order>
        <clientFileId>GPL72SeriesAndSample4Per...</clientFileId>
      </keywordCategory>
      <keywordCategory>...</keywordCategory>
    </keywordOrder>
  </chartDetails>
```

The `xAxis` and `yAxis` elements define labels for the axes and the range of the chart's data. Each `plot` element defines a collection of points to plot and the type of shape to use to plot each point, including its form (circle, triangle, etc.), size, and colors. It also defines a label to put in the chart's legend. Each plot is associated with a different kind of entity, such as a series or a sample. Different plots can use different entity types within the same chart.

The `keywordOrder` element is used to define the categories, keywords, and the order that they appear in the keyword drop-down of the chart. Each `keywordCategory` defines a list of keywords and a file that lists the entities and their associated keywords in a comma-separated file like the following:

```
GSM231232
GSM231233
GSM238440,gcrma
GSM238441,gcrma
GSM238442,gcrma
GSM239567,rma
GSM239568,rma
GSM239569,rma
GSM239570,rma
```

More than one keyword may be specified for each entity.

After the chart details have been retrieved, the client program makes an `XMLHttpRequest` to retrieve information about each of the plot's marker shapes. This information is used to display and interact with the coordinates in the chart. The following is an example of the XML returned by the server:

```
<shapeInfo>
  <name>.</name>
  <size>6</size>
  <centerX>2</centerX>
  <centerY>2</centerY>
  <hitMap>00000001100011110001100000000000000</hitMap>
</shapeInfo>
```

The size element indicates the pixel width and height of the shape. The center coordinates indicate which pixel in the shape represents the shape's center. When a point is projected onto the chart, the displayed shape will have its center pixel aligned on top of the projected chart coordinates. The `hitMap` element holds an array of 1's and 0's indicating which pixels in the

shape's drawn image are considered part of the shape. Usually, bits with a 1 indicate that the pixel's color changes when the shape is drawn.

The chart's coordinate listing is returned from the web server as a comma/newline delimited text file because parsing XML documents becomes very slow when thousands of coordinates are retrieved. The information returned includes an entity's unique ID, name, data x-coordinate, data y-coordinate, and dataset index. The unique ID identifies which entity is associated with each coordinate in the chart. This ID can be passed in a request to the server to retrieve the metadata for the coordinate. The zero-based dataset index specifies to which of the chart's datasets a coordinate belongs to. These datasets are stored in the order in which they are defined in the above chart XML, with a zero index referring to the first dataset in the XML. The following is a sample listing of coordinates for a chart:

```
id,name,x,y,datasetIndex
332,GSM80170,4.322560,1.712690,0
333,GSM80171,4.252000,1.701650,0
334,GSM80172,4.322700,1.698520,0
335,GSM80173,4.318320,1.697340,0
336,GSM80174,4.323020,1.693830,0
337,GSM80175,4.323690,1.715050,0
338,GSM80176,4.310420,1.717220,0
255,GSM81914,4.774970,2.345790,0
256,GSM81915,4.830610,2.285830,0
```

After these XMLHttpRequests have completed successfully, the client-side program is ready to display the chart. There are numerous steps involved in displaying the chart. The chart and axes labels must be displayed, the chart border and tick marks must be displayed, the tick labels must be displayed, and the actual chart image containing the plot of all the chart points must be shown.

Due to the complexities involved in drawing some of the text labels on the screen, such as the sideways drawing of the y-axis label and the power of ten labels at the ends of both axes, the servlet contains a module for creating text label images. Upon receiving a request, it draws text in a variety of fonts, attributes, and orientations into an image and sends this image in the response using the PNG format. The module is also capable of providing metric information so that the client can get access to the width and height of a string of text drawn using the various styles. The servlet sends an XML document such as the following in response to these requests:

```
<textInfo>
   <text>x 10</text>
   <width>23</width>
   <height>10</height>
</textInfo>
```

After receiving this information via XMLHttpRequests, the client-side program can determine precisely where to position labels on the chart.

Once all of this information has been received from the server, the client program is ready to display the chart. The program projects each of the horizontal and vertical tick marks to the

chart's screen space and places a `div` element to create the chart's border and a small, one-pixel wide `div` for each of the tick marks. It then creates `<img>` tags for all of the tick, axis, and chart labels, aligns them on the chart, and sets their `src` properties to point to the servlet's text image module to generate an image containing the appropriate text.

The program also creates one `<img>` tag for each of the chart's dataset markers. These images' `src` properties are set to point to a module in the servlet that generates PNG images containing a drawing of a single marker. The markers are drawn using the highlight color defined in the chart XML. When the user moves the mouse over a point in the chart, the program places this shape image on the chart on top of the point to highlight it.

Finally, the program generates an `<img>` tag to display the plot of all the points. It sets the image's `src` property to point to a module in the servlet which projects all of the chart's points and draws them to an image using the chart's marker shape, size, and colors.

Before relinquishing control back to the browser, the program performs one other significant operation. In order to keep track of which point on the chart the user has moved the mouse cursor over, the program needs to be able to efficiently reverse the projection from the screen coordinates of the cursor to the chart's data points. To achieve this, the client-side program employs an algorithm of recursively dividing the chart's screen space into quadrants, creating a tree structure, with each node containing 0 or 4 child quadrants. These quadrants are divided further and further until the child quadrants contain 20 or fewer points in them. This tree of quadrant nodes is generated during this stage of the program's execution.

4. Interacting with the chart

After the chart has been displayed, the user is then able to begin interacting with the chart. The following sections describe the processing and server communication that occurs as the user works with the chart.

4.1 Selecting points and viewing metadata

When the "Select" toolbar button is selected, the user is able to select and deselect points in the chart. The user selects points by dragging a box with the mouse around the points in the chart and releases the mouse button. When this occurs, the program iterates through all of the chart's points and determines which points are in the interior of the dragged rectangle. These points are then added to the selection. Points can be deselected by holding the CTRL key and dragging a box around the points. The points on the interior of the dragged rectangle will then be deselected. The selection can be cleared by right-clicking in the chart and clicking the "Deselect All" menu item.

Anytime the selection changes, the chart must be updated to reflect the new selection. Points that are in the current selection must be displayed in a different color than the other points. The client-side program accomplishes this by overlaying the original background plot image with selection images. These overlaid images display the selected points using the selection color and

hide the markers drawn in the original color beneath them. These selection images are updated every time the selection changes.

The mechanism is different for generating the selection images than it is for generating the original chart plot image. When the original plot image was generated, the server-side servlet iterated through all of the points in the chart and projected and drew them onto the image. For the selection images, however, the projection of the points occurs on the client-side. The projected screen coordinates are then encoded into a form which is transmitted to the server as part of the images' source URLs. This strategy moves the burden of projecting all of the chart's points to the client side; the server is not required to iterate through all of the points within the chart every time the selection on the client changes. Instead, it simply reads the coordinates from the encoded URL data and then draws the appropriate markers to an image. Due to the limits in the size of URLs (about 2,048 characters in IE), when a large number of points are selected, not all of the encoded coordinate data can fit into a single URL. Thus, the client program may have to generate multiple images to cover the entire selection. The limit is set at 540 points per image.

In addition to displaying all of the selected points in a different color, the client program also adds all of the selected points into a navigable tree. This tree displays up to 100 selected points at a time. The user can view all of the points by using navigation controls above the tree. When the user clicks on a point in the tree, the node is expanded and displays metadata associated with that point. As with the browser tree, the metadata for the points in the tree is downloaded dynamically only after the node is expanded. This metadata is downloaded from the server through an `XMLHttpRequest` only after the user expands the node rather than downloading the metadata for every selected point when the selection changes. The following is an example of the metadata XML document returned by the server for a point:

```
  <metadata entityId="123">
    <item key="Title" link="">Sample WT(2)</item>
    <item key="SampleID"
              link="http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSM4559
        3">GSM45593</item>
    <item key="SeriesID"
              link="http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE2422
        ">GSE2422</item>
    <item key="PlatformID"
              link="http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GPL72">
              GPL72</item>
    <item key="Description" link="">Heads from young flies (2-4days) were
              collected at ZT16 after 3 days of training in LD
              conditions.</item>
    <item key="Characteristics" link="">[none]</item>
    <item key="Data-Processing" link="">NaN</item>
    <item key="Value Description" link="">
              Exp (Base2) of the expression value obtained using GCRMA</item>
</metadata>
```

Each item consists of a `key` field name and a value. Each item may also have a `link` associated with it. Items that contain a `link` in the XML will display the link in the metadata tree on the page. When the user moves the mouse over a node in the metadata tree, the associated point in the chart is highlighted by overlaying the highlight marker image on top of the point. (This

image was created when the chart was initially displayed. See the previous section for additional information.)

Metadata is also displayed when the user moves the mouse cursor over a point in the chart. When the mouse is moved over the point, the point is highlighted. After the mouse has remained positioned over the point for 500 ms, an XMLHttpRequest is initiated to download the metadata for the point. When the request completes, the metadata is displayed on the screen in a tooltip. The 500 ms delay helps to prevent the server from becoming overloaded with requests for metadata.

4.2 Zooming

When the "Zoom" toolbar button is selected, the user may zoom in or out of the chart. The user zooms in by dragging a box over a rectangular region in the chart. The region occupied by this box when the mouse button is released becomes the new view in the chart. The user may also zoom in by right-clicking within the chart and clicking the "Zoom in (x4)" menu item, which will zoom the view by a factor of 4 and center the view over the data point where the right-click occurred. The user zooms out by right-clicking within the chart and clicking the "Zoom out" menu item, which will zoom the view all the way out to display the entire data space of the chart.

The user can also go to the previous zoom coordinates by right-clicking in the chart and selecting the "Previous zoom" menu item. The client program maintains a list of the 10 previous zoom coordinates.

Every time the zoom coordinates of the chart change, several portions of the chart are updated. First, the chart's plot background image must be updated to the new coordinates. A new image is generated and its `src` property set to point to the servlet's image generation module. URL query parameters containing the new zoom coordinates are supplied. Second, the chart's tick marks are updated. The previously drawn tick marks are erased. If the view is zoomed in, new tick marks are dynamically generated and added to the original tick marks which were declared in the chart XML, and these ticks are drawn to the screen using Canvas. The labels for these ticks are created, aligned, and added to the DOM. Third, the selection images are updated. All of the previous selection images are removed from the DOM and new images containing the new projected coordinates of the selected points are added.

4.3 Saving and loading selections

The user can save or load selections to their local hard disk by right clicking in the chart and selecting the "Load selection" or "Save selection" menu items when the "Select" toolbar item is selected.

JavaScript code running within web pages is not given any access to the user's file system. The only means of loading and saving files is to redirect the content to be saved or loaded through the server. In addition, loading and saving files cannot be accomplished by simply using XMLHttpRequests. Uploading a file to the server requires that an HTML form with the selected file be submitted to the server. Downloading a file requires that the browser be redirected to a new page to receive the download. The charts program accomplishes these tasks by making use of HTML inline frames.

The save selection operation uses a hidden <iframe> on the page to induce the browser to download a file. The iframe is loaded with an HTML page that contains a form with a hidden control. When a save operation is initiated, the program creates a text variable containing all of the IDs of the selected points. The program then assigns this text value to the value property of the hidden control. The program then submits the form in the iframe, which initiates an HTTP POST to the web server. The servlet receives the request and reads the text content. It then writes the contents to the response output and sets some response HTTP headers indicating that the data is to be interpreted by the browser as a downloadable file. The user can save this file to the file system.

Selection load operations also make use of an <iframe>, but unlike with the save operation, this iframe is actually displayed to the user. This iframe loads an HTML page that contains a form with a file input control. The user uses this control to browse to and select a file on their file system. The user then clicks the open button to load the file. The button is actually a submit form control that performs an HTTP POST, uploading the selected file to the web server. The server reads the uploaded contents and then simply writes the content to the response output, with a small amount of HTML encoding. When the iframe has finished loading the response, the client program reads the content and parses out the selection information. The program then modifies the selection, generates the appropriate selection images, and updates the metadata tree.

4.4 Exporting data

GPLBrowse allows the user to export a customized collection of data based on the current selection of points. The chart XML that is generated during the offline data preprocessing can contain descriptions of exportable collections of data. Each data collection consists of a series of datasets. The rows and columns within the datasets can be mapped to the entities that correspond to plotted coordinates on the charts, although this is not a necessity. For instance, the columns in a dataset may correspond to NCBI GEO samples. The mapping may also be indirect. For instance, the columns in the datasets may correspond to NCBI GEO series. Samples can then be mapped to a dataset column by a series that contains them. When the rows and/or columns in the datasets are mapped to entities, the first row and/or column in each dataset must provide the names of the mapped entities. The full datasets, on the other hand, are required to be mapped to entities. The names of the mapped entities are used as the filenames identifying the datasets.

When the user has selected some coordinates on the displayed chart, a menu item is displayed that allows the user to initiate an export. When this menu item is clicked, the client JavaScript opens a dialog box allowing the user to select the type of data to export. When the user clicks the export button, the export operation is executed. First, the client program decides which data needs to be exported using the currently selected points on the chart as a basis. The points are mapped to the rows and/or columns in the datasets, and only the matching rows and/or columns are selected for export. For example, assume that the columns in the exportable datasets represent NCBI GEO samples, and the points in the displayed chart also correspond to NCBI GEO samples. When the export is executed, the currently selected samples on the chart are used to select individual columns within the datasets. When the server downloads the data to the

client, it only includes the selected columns in the output. Datasets that do not contain any selected rows or columns are not included in the export data.

The chart points may also be used to select entire datasets. For instance, if a chart's points correspond to NCBI GEO series and the datasets in an export also correspond to GEO series, then the points currently selected in the chart are used to select complete datasets to be exported. All rows and columns in the selected dataset are included in the export.

It is also possible for both of these scenarios to apply at the same time. For instance, consider a chart that plots both series and samples together. The samples select columns in the exportable datasets while the series select complete datasets. In this situation, the selection of entire datasets will supersede the selection of individual rows and columns. Thus, if a selected series maps to a dataset in the export, the entire dataset will be exported even if only a few of the samples in the dataset are selected.

Once the client has determined what data must be exported, a request is sent to the server to compile this data into a compressed archive file and download it to the client. The client lists the mapped entities in the post data of the request, so that the server will know which data must be included in the download. The URL query string cannot be used for this purpose due to the fact that it may not be long enough to list all of the mapped entities. As is the case when downloading files listing the current selection (described in the previous section), the client must make use of a hidden HTML `iframe` to send this request and initiate the export download.

5. Integration of the Yahoo User Interface Library (YUI)

Designing complex web user interfaces is very difficult. Care must be taken to ensure that the HTML code that is used is compatible with all of the targeted web browsers. However, browsers can render certain types of HTML very differently. It can take a lot of time and effort to find ways to write HTML that renders in a desirable fashion on all of the browsers. Wherever possible, it is a good idea to make use of HTML and JavaScript that has already been designed and thoroughly tested in all of the major browsers. This is what makes the Yahoo User Interface library, or YUI, a great asset to web application developers. YUI provides an abundance of widgets, such as buttons, tree views, and dialog boxes, that are targeted specifically for Ajax developers. Much of the user interface in GPLBrowse is made from these widgets.

5.1 Layout managers

GPLBrowse makes use of the YUI Layout Manager, a recent YUI addition, to control the layout of the different sections of the application. At the top level, GPLBrowse creates one layout manager to separate the chart browser tree from the main area where the charts are displayed. These divided segments are referred to as "layout units" in YUI. When the user displays a chart, more layout managers are created and added to the top manager in a hierarchical fashion to further divide the chart area and create more layout units.

The nicest feature that the layout manager provides to the application is its ability to handle resize events in the browser. When the size of the browser window changes, the layout managers automatically adjust the size of their layout units. In addition, the layout managers allow the user to grab the layout unit borders and resize them with the mouse. When the layout

manager resizes one of its units, it notifies the main program by invoking a callback function. The program can then respond by manipulating the HTML and controls inside the layout unit(s). In GPLBrowse, when the program is notified that the layout unit containing the chart has been resized, the program responds by resizing the chart, regenerating the tick marks, and realigning all of the chart's labels. Resizing the chart requires that all of the chart's images – the main plot and the selection, search, and keyword images – be regenerated with the new chart dimensions. GPLBrowse enforces a minimum size restriction on the chart section of the page. This minimum size is configurable in an XML configuration file. If the layout unit is too small to display the entire chart section, scroll bars are displayed to allow the user to see the entire chart.

Another feature of the YUI layout manager allows the user to minimize and restore a layout unit by clicking a button in the unit's title bar. When minimized, the content of a layout unit is pushed off of the screen so that the unit uses a minimal amount of screen space. Both the chart browser tree and the chart legend layout units can be minimized in GPLBrowse so that the chart can be allocated a larger amount of screen space. The browser tree is automatically minimized when a chart is displayed.

5.2 Menus

Another type of widget that is used prominently in GPLBrowse is the YUI menu. GPLBrowse relies on YUI to create both its main menu at the top of the window and the chart context menus that are displayed when the user right-clicks within the chart. The contents of both of these menus changes depending on the state of the application.

It is easier to change the contents of the context menu because it is generated at the moment it is needed, and the items may be added to the menu depending on the state of the application – such as which toolbar item is selected or whether any points are selected in the chart or not. The main menu is a little more difficult to manage because it is always displayed and cooperation is needed to ensure that menu content is inserted and removed at the appropriate locations. A MenuManager class was created to be responsible for managing the locations where content is inserted and re-rendering the menu when necessary. When this class is first instantiated, a number of top-level menu items are added to the menu – GPLSuite, File, and Help. Content is added to these top-level menu items by using what is referred to as menu "insertion points." The insertion points are added to the top menu in a certain order. Each is given a descriptive name that can be used to get access to the insertion point. When the contents of an insertion point needs to be recreated, it can be cleared, and all of its menu items can be re-added. After the new content has been added (or simply removed), the menu manager is told to re-render the menu. When the menu is rendered, the content of each insertion point is added to the menu in the order that the insertion points were originally added to the menu. Using this approach, new menu content does not simply get added to the bottom of the existing menu; instead, content can be added in the middle or top of the menu in a predetermined location.

Some base menu items are added to the menus in the beginning and never change. For instance, the GPLSuite links are added to the GPLSuite menu and a number of help items are added to the help menu. This content is never removed from the menu. However, other content is added to the File and Help menus when a chart is displayed, when the browser tree is minimized or restored, and when points are selected in the chart.

5.3 Buttons, dialogs, and tooltips

A number of other miscellaneous items from YUI are used in GPLBrowse. Buttons are used throughout the application. In some cases, simple HTML buttons are used. This is generally the case when the only functionality needed from the button is for the program to be notified when the button is clicked and advanced styling for the button is not desired. In cases where more advanced functionality is needed or a nicer appearance is required, YUI buttons are used. For instance, YUI buttons are used for the Select and Zoom toolbar buttons. These buttons make use of the radio button feature in YUI, in which one and only one of the buttons is selected at a time.

GPLBrowse also makes use of YUI dialog boxes in a number of places. These dialog boxes are implemented as HTML panels that are displayed in front of the other content on the page. While the dialog is displayed, the user can be prevented from accessing the rest of the content until the dialog is closed. The export functionality in GPLBrowse is presented to the user using a YUI dialog. A YUI dialog is also used to prompt the user when one of the GPLSuite menu items is selected and the browser must navigate to a new page. The prompt allows the user to decide whether to open the new web page in a new window, in the current window (erasing their chart), or to cancel the operation.

GPLBrowse also uses YUI tooltips to display instant information to the user. These tooltips are used to display metadata when the mouse hovers over a point for a short period (as described in the section “Selecting points and viewing metadata”). It is also used to display descriptive information about the chart when the mouse is moved into the margins of the chart. Tooltips are also used to display longer descriptions of the folders and charts in the chart browser tree.