Amber Development Guide

Table of Contents

Introduction

What is Amber?

Amber is a powerful Client/Server system with the following characteristics:

- It allows remote clients to use browsers to run Applications with a small download.
- The users are presented with all the normal user interface elements: Buttons, text controls, menus, multiple windows etc.
- Amber is Java based and therefore will run on most operating systems and browsers.
- Amber development is almost identical to normal Java application development.
- Browser HTML limitations are transcended by the Amber system.

Concepts

An Amber system is composed of two sets of systems. One on the client and the other at the server.

- The client system handles the display requirements and interacts with the user. This is in the form of visual display and detecting the various user generated events.
- The server system contains the program logic and any associated business rules.

Communication between the two systems is handled by the Amber Application Transfer Protocol (AATP).

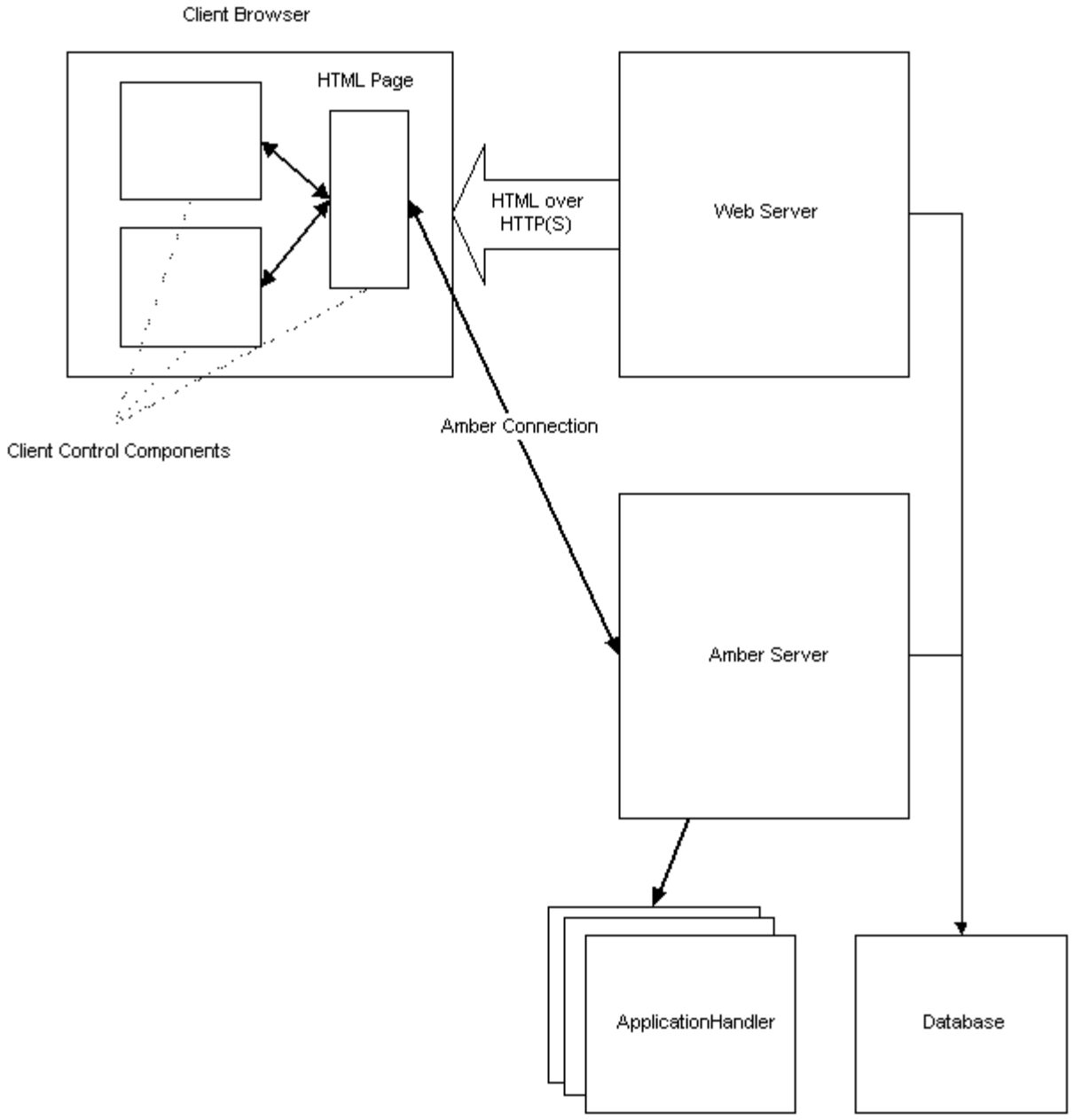
The Amber Client

The Amber client(s) run on the client computer in the target browser. The Amber clients are one or more Java applets which reside in an HTML page in the browser. The client connects directly to the Amber server and establishes a unique session with the server. The client then responds to commands from the server and issues events back to the server as generated by the client clicking or otherwise interacting with the displayed controls. It contains a repository of available Amber controls for use by the Amber server. These controls include all the standard user interface elements such as buttons etc.

The Amber Server

The Amber Server is a stand-alone Java application running on the server computer. It detects client connection requests, identifies the correct application to create for a connection, creates the application, then connects it to the client and starts it running. Once this occurs the application acts in an almost identical manner to a normal Java application. The server supports handling multiple simultaneous applications at once.

The server acts as an application runner. The correct application (a class derived from ApplicationHandler) is identified when the client connects. This class is instantiated and the server connects the socket stream to it and starts the application running by calling the **start** method in the ApplicationHandler. Once the application is running it functions in a similar manner to a normal stand-alone application. It is this ApplicationHandler which gives the client side controls functionality and defines the program logic behind the client display.



Building Applications with Amber

General Overview

When creating an Amber application it is important to understand how the Amber system is to be used. This will help define both the display elements at the client and the required program logic at the server to make the application function correctly. There will be two sides to an Amber application: the client side which is partly HTML and partly Java, and the server side which generally appears as a slightly modified Java application. In general, a good starting point is to view the client as displaying the visual components and the server as running the Java application. The HTML pages provide a structure inside which the visual components will fit.

We will now consider each of these elements in turn.

HTML Pages-Client Components

The client part of Amber sits inside an HTML page. To the HTML page Amber controls appear to be standard Java Applets. The HTML page is formatted for appearance in the normal way. Applet tags are inserted in location(s) where the controls are required. Parameters in the tags define the type of control and the ID of the control. The possible controls range in complexity from the simple which are buttons or labels to the complex which are panels of controls. To some degree the client browser treats them in the same way.

Amber allows multiple controls to exist in an HTML page. ID numbers are used to distinguish between the various controls. HTML page controls may have ID numbers ranging from 0 to 199. There *must* always be an Amber control with an ID of 0. This is the *master* control and handles the requirements of messaging to the server. Any Amber control type may be a master. Control ID's may not overlap. If two controls share the same ID then the behaviour of the Amber client system is undefined. As Amber allows container components such as panels (see below) it is not absolutely vital to have multiple controls within an HTML page. The panel allows a single component to embed multiple visual controls thereby creating a simple, fully viable Amber application.

We will now look at a typical Amber applet tag and describe some of the salient features.

Amber Applet Tag

The following is a typical Amber applet tag which is inserted into the HTML page:

```
<APPLET
  ARCHIVE = "AmberClient.jar"
  CODE    = "amber.client.RBase.class"
  NAME    = "MainPanel"
  WIDTH   = 640
  HEIGHT  = 480
  HSPACE  = 0
  VSPACE  = 0
  ALIGN   = middle
>
<PARAM NAME = "Component" VALUE = "amber.client.panel.BasePanel">
<PARAM NAME = "ID" VALUE = "0">
<PARAM NAME = "SERVER" VALUE = "127.0.0.1">
<PARAM NAME = "Port" VALUE = "21384">
<PARAM NAME = "PAGEID" VALUE = "2000">
<PARAM NAME = "EVENTMASK" VALUE = "1F">
<PARAM NAME = "Extension0" VALUE = "Name|John">
<PARAM NAME = "Extension1" VALUE = "Job|Programmer">
</APPLET>
```

The applet tag at the top is fairly normal for an HTML Java applet. The following fields are displayed in our example although all the standard properties can be used:

- **Archive**. This is the name of the archive jar which contains the Amber Java client classes.
- **Code**. This is the name of the primary applet derived class which is to be started. For most Amber clients this will be **amber.client.RBase**.
- **Name**. This is the name which is assigned to this applet. This is rarely used and is only relevant where Javascript is used to communicate with the applet.
- **Width**. This is the width of the applet in the HTML page.
- **Height**. This is the height of the applet in the HTML page.
- **Hspace**. This is the amount of horizontal space to add around the outside of the applet.
- **Vspace**. This is the amount of vertical space to add around the outside of the applet.
- **Align**. Where in the allocated space for the applet will the applet appear.

The following parameters are unique to Amber. They are used to configure the Amber control and vary depending on the control displayed.

These are some of the common Amber parameters:

- ID. This is the ID number of the Amber control. This ID number is used to link the client control with the server ComponentHandler. These numbers must be unique and there must always be a control with an ID of 0. The ID 0 control is required to have several other parameters which relate to its role as master control.
- Component. This is the client class which is the display object. In this case the display object is a panel (amber.client.panel.BasePanel) however any of the normal components in amber.client.panel can be used.
- Server. Optional, only used by ID 0. This defines the location of the server to which

the Amber client system will connect. This may be either an IP (Internet Protocol) address or a standard dotted notation address such as `amber.clearfield.com`. If no address is specified then the base address of the server in the HTML URL is used.

- PageId. Required, only used by ID 0. This integer is used by the Amber server to identify the type of application to be attached to this client.
- PageSubId. May be optional, only used by ID 0. This integer is used to further refine the type of application to attach. The server can be configured to ignore this integer when identifying the correct application. In this case the number may be ignored or can be used to pass additional information to the starting ApplicationHandler at the server.
- ExtensionX. Where X is a monotonically increasing integer starting at 0 (i.e. Extension0). Optional. This is a series of properties which are to be passed back to the Amber Server by the the ID 0 component. These extension properties can be read from the ApplicationHandler using the getRemoteProperties() function. The value of the Extension param is in the form "name|value" where name is the name of the property and value is the value given to the property.

Refer to Appendix: Common Amber Applet Tags, for a more detailed list of parameter tags common to Amber components.

Generally each specific component has additional parameters which depend on the control which is displayed.

Applications

Applications are the classes which are run in response to an incoming client connection request from the master (ID 0) client applet. They all derive off the base class:

`amber.server.application.ApplicationHandler`.

The Amber server resides at the server machine. For small servers this will be the same computer as the Web Server computer. The connection process is as follows:

- The Amber server waits for incoming connections on a particular server port. The default value is 21384, although this is configurable.
- When an incoming connection from the Amber client master control is received the Server reads the Page ID and Page Sub ID fields transmitted by the client.
- The Server looks in the Amber configuration database and matches the ID's to the corresponding ApplicationHandler.
- The ApplicationHandler extended class is created using its default constructor.
- Initial connection information is used to configure the ApplicationHandler. This includes the Page ID and the Page Sub ID fields.
- The start function on the ApplicationHandler is called. This finished the ApplicationHandler initialisation and creates the required messaging threads. Any required user initialisation is also performed in this function.

In general, the ApplicationHandler reflects the user interface specified in the HTML page. The client controls are associated with ComponentHandlers which are used to alter the state of the client controls and process any events transmitted.

An ApplicationHandler corresponds to a standard Java Application.

- It handles all the requirements of the AATP to message between the server application and the corresponding Amber client controls.
- A server application **extends** the `amber.server.application.ApplicationHandler` class to create a new application.
- An ApplicationHandler contains a number of *ComponentHandler* derived control handlers to control Amber client controls. These correspond to the normal Java AWT controls.
- An instance of the ApplicationHandler derived class is created for each incoming Amber connection. Once the ApplicationHandler is instantiated it acts as a stand-alone application.

A simple application which corresponds to the previous HTML tag example would look as follows:

```
package untitledPackage;

import java.net.* ;
import java.io.* ;

import amber.type.server.*;
import amber.server.application.*;
import amber.client.RConstants ;

/**
  * This is the Application Handler.
  *
  * @author Insert your name here
  * @version 1.0.0
  * @see amber.server.application.ApplicationHandler
*/
public class SimpleApplicationHandler extends ApplicationHandler
{
    /**
      * Declare any panels or frames this application uses.
      * Note that each panel/frame corresponds to one APPLET tag in the
      * HTML.
      * Also note that the panel/frame ID must match the ID parameter in
      * the html.
    */

    /**
      * Here is the first panel in this sample application
    */
    private GenericPanel pnlMain = new GenericPanel (0, this) ;

    /**
      * Default constructor
    */
```

```
public SimpleApplicationHandler () throws IOException
{
    super (0, RConstants.InvalidPageSubId) ;
    defineComponents ();
}

/**
  * The initialising constructor.
  * @param appIdentifier. An int uniquely identifying this instance of
  * application.
*/
public SimpleApplicationHandler (int appIdentifier) throws IOException
{
    super (appIdentifier, RConstants.InvalidPageSubId) ;
    defineComponents ();
}

/**
  * The initialising constructor.
  * @param appIdentifier. An int uniquely identifying this instance of
  * application.
  * @param newConnection. The Socket which is connected to the page in
  * operation.
*/
public SimpleApplicationHandler (int appIdentifier, Socket
  newConnection) throws IOException
{
    super (appIdentifier, RConstants.InvalidPageSubId, newConnection);
    defineComponents ();
}

/**
    * Do initialisation that needs to be done within the constructor,
    * eg add any panels/frames this application uses.
    * Use "add" to add the component into the application
    * messaging loop.
*/
protected void defineComponents ()
{
    // Add any panels/frames the application will use here.
    add (pnlMain, new XYConstraints (0, 0, 640, 480));
}

/**
  * This function initiates the functioning of the ApplicationHandler.
  * This function is required as the application handler will not
  * immediately start operation until it is handed the socket by the
  * main handling system.
  * @param newConnection. The Socket which is connected to the page in
  * operation.
  * If null uses the connection already set.
  * @exception java.lang.IllegalThreadStateException containing any
  * problems.
*/
public synchronized void start (Socket newConnection) throws
  IllegalThreadStateException
{
    super.start (newConnection);
    setUp ();
}

/**
    * This function is called when the program first starts.
```

```
        * Insert into here any initial set up code you may require.
    */
    public void setUp ( )
    {
        try
        {
            // Add any setup logic required here
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

We will now consider the various parts of the ApplicationHandler in turn.

The application class is called SimpleApplicationHandler. It derives off the class amber.server.application.ApplicationHandler. The ApplicationHandler base class contains the logic required to link the server application to the remote client display.

The class which controls the client RPanel is defined in the line:

```
    /**
      * Here is the first panel in this sample application
    */
    private GenericPanel pnlMain = new GenericPanel (0, this) ;
```

The GenericPanel class is a class which extends a hierarchy of classes ultimately extending the ComponentHandler class. This class controls the remote client applet and handles the specific requirements for the server to remotely control the client display for that panel. The constructor for this class contains two arguments these are:

- The component ID. This *must* correspond to the ID defined in the applet parameter tag. In this case there is only one component and its ID is 0.
- ApplicationInterface handle. This is a handle which points to the Application which controls the ComponentHandler. The class ApplicationHandler implements the ApplicationInterface for you.

The SimpleApplicationHandler has a variety of constructors but *must* have a default constructor with no parameters. It is this constructor which is used when the Amber Server instantiates the class.

```
    /**
      * Default constructor
    */
    public SimpleApplicationHandler () throws IOException
    {
        super (0, RConstants.InvalidPageSubId) ;
        defineComponents ();
    }
```

The constructor calls the super constructor for ApplicationHandler and then calls the

defineComponents function. This function is required to define the components which will message to the remote client. It is an abstract function in ApplicationHandler and must be supplied by the deriving class.

```
/**
    * Do initialisation that needs to be done within the constructor,
    * eg add any panels/frames this application uses.
    * Use "add" to add the component into the application
    * messaging loop.
*/
protected void defineComponents ()
{
    // Add any panels/frames the application will use here.
    add (pnlMain, new XYConstraints (0, 0, 640, 480));
}
```

This function contains one function add which is used to tell the ApplicationHandler that this panel exists at the remote client and that its physical characteristics are x, y location 0,0 and size width=640, height=480. The physical characteristics are not absolutely required for remote Browser initiated connections however they are vitally important for remote Application initiated connections.

In general, the Amber Server will create the ApplicationHandler extended application using the default constructor. When the class is instantiated it is not connected to the remote client. When the ApplicationHandler is connected and all the required properties are set the start function is called in the ApplicationHandler. This function is responsible for the final initialisation of the ApplicationHandler and any user required initialisation which requires that the application is actually connected to the remote client. The function typically looks as follows:

```
/**
  * This function initiates the functioning of the ApplicationHandler.
  * This function is required as the application handler will not
  * immediately start operation until it is handed the socket by the
  * main handling system.
  * @param newConnection. The Socket which is connected to the page in
  * operation.
  * If null uses the connection already set.
  * @exception java.lang.IllegalThreadStateException containing any
  * problems.
*/
public synchronized void start (Socket newConnection) throws
  IllegalThreadStateException
{
    super.start (newConnection);
    setUp ();
}
```

It is vitally important that the first line in the start function is the call to the super.start function. If this is not called then the ApplicationHandler will not be correctly initialised. Once this line is complete then any required user initialisation may then be performed. In this example, the SimpleApplicationHandler calls an initialisation function called setUp.

In our example the setUp function does nothing however in normal circumstances it is here that the final set up of the ApplicationHandler is performed. Any processing which requires that the ApplicationHandler is properly initialised and that the client is connected to the ApplicationHandler should be placed here. For example, a call to the getClientScreenSize() function requires a connection to be established. This function call must be placed in setUp and not in defineComponents() or the class constructor. For more information on active parameters see the later section Active Parameters.

```
    /**
        * This function is called when the program first starts.
        * Insert into here any initial set up code you may require.
    */
    public void setUp ( )
    {
        try
        {
            // Add any setup logic required here
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Exception Handling

It is worth noting the amber.server.exception structure. All Amber exceptions derive off the class amber.server.exception.AmberException. This exception can be instantiated with another exception. This allows nested chains of exceptions to be created. This reduces issues with catching and re-throwing exceptions where the stack trace of the original cause is lost. Calling printStackTrace() on an AmberException (or AmberException extended exception) causes *all* causal exceptions to be dumped in sequence from newest to oldest. Thus the last exception printed is the original exception generated.

Server Components (ComponentHandler's)

If the ApplicationHandler corresponds to a standard Java application then the ComponentHandlers correspond to the standard Java AWT controls. They are designed to closely mimic the standard Java version 1.1 AWT controls to facilitate ease of understanding and use. All server components ultimately extend the amber.server.component.ComponentHandler class. It is this class which handles the command and event processing required to allow the remote Amber controls to function. Amber contains a large range of available components in the amber.server.component package including:

- Buttons. `amber.server.component.ButtonHandler`. This is the standard button.
- Chart recorders. `amber.server.component.TimePlotHandler`. This is a system which acts as a chart recorder displaying a range of values over time. When the chart recorder is full it scrolls to remove the left most value.
- Check boxes. `amber.server.component.CheckboxHandler`. This is a check box control allowing the user to select a boolean true/false value.
- Check box panels. `amber.server.component.CheckboxPanelHandler`. This is a panel containing a number of radio state buttons. This allows the user to select an item from an available range of options.
- Choice controls. `amber.server.component.ChoiceHandler`. This is a drop-down list of options. The user may select any of the items in the list.
- Complex list controls. `amber.server.component.ComplexListHandler`. An extension to the standard list. This control allows multiple columns, parent/child lines, and images.
- Float Buttons. `amber.server.component.FloatButtonHandler`. This is an extended button with the following features: display of an image in the button as well as a caption, may be configured to toggle state (allowing the button to act as one of several possible choices).
- Images. `amber.server.component.ImageHandler`. This control allows the display of images. The types of images which can be displayed are JPEG and GIF.
- Image button. `amber.server.component.ImageButtonHandler`. This is a button composed of two images: an up image which is the default and a down image when the button is pressed. Apart from this the image button acts in the same manner as a standard button.
- Labels. `amber.server.component.LabelHandler`. This is a static control which can display a single line of text.
- Lists. `amber.server.component.ListHandler`. The list is a control which displays a list of text.
- Menu Bars. `amber.server.component.MenuBarHandler`. This component functions almost identically to the standard Java menu bar control. It may only be attached to Frame windows.
- Pop-up menus. `amber.server.component.PopupMenuHandler`. This menu may be used on any panel or frame component. It allows pop-up stand-alone menus to be created.

- Text Areas. `amber.server.component.TextAreaHandler`. This is a multi-line text entry control.
- Text Fields. `amber.server.component.TextFieldHandler`. This is a single line text entry control.
- Tree Lists. `amber.server.component.TreeListHandler`. This is a control allowing parent/child data to be displayed in the form of a hierarchical tree of nodes. Both text and images may be displayed at each node.
- Wrapping Text Areas. `amber.server.component.WrappingTextAreaHandler`. This is similar to the TextAreaHandler however this control can have pre-set margins within which the text is forced to wrap (move to the next line).

ComponentHandlers are instantiated for use in the ApplicationHandler. A ComponentHandler may be instantiated anywhere in an ApplicationHandler or panel (see below). In our previous example the SimpleApplicationHandler instantiated a GenericPanel type of ComponentHandler.

ComponentHandlers are instantiated in one of two ways. These are:

1. The ComponentHandler handles a control which will reside inside a panel at the remote client (see panels section below). In this case there is no corresponding applet tag in the HTML page for this component. In this case the Amber Server will automatically assign an ID for the component and this information is not required. This form of the ComponentHandler takes one argument which is:
   a. ApplicationInterface. This is the handle to the ApplicationHandler within which the ComponentHandler resides.
2. The ComponentHandler is to control a remote component specified directly in an HTML applet tag. In this case the ID is prespecified in the applet tag and cannot be changed. For this reason the ComponentHandler takes two arguments:
   a. ID: an integer which corresponds to the ID number specified in the applet tag. Should these not correspond the ApplicationHandler will not be able to control or receive events from the remote component.
   b. ApplicationInterface. This is the handle to the ApplicationHandler within which the ComponentHandler resides.

Typically the second form of the ComponentHandler is only ever used within the extended ApplicationHandler class itself. Inside panels and frames the first form of the ComponentHandler constructor is used.

Thus to create a ButtonHandler for a panel the following line could be used:

```
private ButtonHandler btnTest = new ButtonHandler ( this ) ;
```

To instantiate a ButtonHandler to control a remote button component with an ID of 6 specified in the HTML applet tag the following line could be used:

```
private ButtonHandler btnTest = new ButtonHandler ( 6, this ) ;
```

In both cases it is assumed that the ComponentHandler was constructed from within an ApplicationHandler extended class. Should the ComponentHandler be constructed with an Amber panel the constructor changes. The required Application Handler is found using a panel member function getParentApplication() which returns a handle to the parent ApplicationHandler (see panels). The new form would be

```
private ButtonHandler btnTest = new ButtonHandler ( getParentApplication()
) ;
```

Once instantiated the component must be added to the application or panel. This is done using the add function. The syntax of this function is the same in both ApplicationHandlers and panels and is:

```
public void add ( ComponentHandler newComponent, Object constraints ) ;
```

where the arguments are:

- newComponent. The ComponentHandler to add to the ApplicationHandler or panel.
- constraints. An Object with the constraints to use when creating the component. At the current point in time this object must be of the type `amber.type.server.XYConstraints`. This is the only constraint class understood by the Amber system. Use of other standard Java constraints will be rejected.

For example:

```
add ( btnTest, new XYConstraints ( 10, 10, 80, 40 ) ) ;
```

The ComponentHandler is now ready for use. Each ComponentHandler has a set of standard functions available for use and a set of extension functions which correspond to the normal functions available in a normal AWT control. The base ComponentHandler class is responsible for messaging and handles all the extended requirements for event reception and processing.

Event Handling

Event handling in Amber is almost identical to that of normal Java AWT. The following events are supported:

- Action Events.
- Component Events.
- Focus Events.
- Item Events.
- Key Events.
- Mouse Events. This includes both standard mouse and mouse motion events.
- Text Events.
- Window Events. This is a special case see below.

With the exception of Window Events the events thrown are the standard Java AWT event classes. This allows the normal addListener interface corresponding to the required event to be used when listening for events. *All* ComponentHandler derived classes can generate any of these events. In some cases although the event interface exists no event will ever be sent by the corresponding control. For example, a ButtonHandler will fire Action events but not Item events.

All ComponentHandlers may fire the following events:

- Component Events.
- Focus Events.
- Key Events.
- Mouse Events.

Window Events are a special case in Amber. The Window event used in Amber is an instance of the class `amber.awt.event.ComponentWindowEvent`. The corresponding listener class is `amber.awt.event.ComponentWindowListener`. These classes are designed to appear similar to the corresponding event and listener in standard Java.

An alternative mechanism for handling events is available when extending existing ComponentHandlers. By overriding the processEvent functions it is possible to handle incoming events without resorting to the listener interface. Comparing this mechanism to that available in standard Java applications the programmer must enable the events which are to be processed by calling the enableEvents function with the events to handle. Amber handles this in a similar fashion however the function to enable events in a ComponentHandler is:

```
void setEventMask ( int eventMask )
```

Where the eventMask parameter is one or more of the Mask constants specified in `amber.client.RConstants`. These mask values correspond to the types of events which the ComponentHandler may be expected to process. The event mask values can be seen in Appendix: Common Amber Applet Tags.

Specifying the Events Sent over the Network

Normally the programmer will not be required to directly alter which events are transmitted by the Amber system. Just specifying and calling the addListener functions defines which events are transmitted from client to server. However there will be occasions when a more granular definition of exactly which events are transmitted from the client is required. The normal reason for this would be to reduce network traffic however we have seen in the previous section that extending ComponentHandlers may require special event handling. In these cases the default event behaviour of Amber can be overridden for a ComponentHandler by calling the function:

```
void setEventMask ( int eventMask )
```

Where the eventMask parameter is one or more of the Mask constants specified in

`amber.client.RConstants`. This function specifies both at the client and server which events will be transmitted from the client component to the corresponding ComponentHandler.

For example adding a key listener using the following code line:

```
component.addKeyListener ( this ) ;
```

to a component would enable the following:

- Key Pressed Events.
- Key Released Events.
- Key Typed Events.

Often only one event would be required, Key Released for example. To override the default behaviour and only allow Key Released events the following code segment could be used:

```
component.addKeyListener ( this ) ;
int mask = component.getEventMask ( ) ;
mask &= ~(RConstants.KeyTypedEventMask |
      RConstants.KeyPressedEventMask) ;
component.setEventMask ( mask ) ;
```

The addKeyListener function enables all three events and the following two lines directly disable Key Typed and KeyPressed for this component. It is important to note that all other program code will also no longer receive these events from this component.

An alternative to the above code could be used in the case where an extended class wishes to alter the events handled. This is by using the protected functions:

```
addEventMask ( int eventMasks ) ;
removeEventMask ( int eventMasks ) ;
```

This allows selected event types to be added or removed from the events which will be created.

Basic Functions

All ComponentHandler extended classes are capable of certain generalised operations useful to programmers. These operations include:

- public void forceComponentGetUrl (String urlString). This function forces the remote client browser to get an HTML URL. This would override the current page the Amber client resides in.
- public void forceComponentGetUrl (String urlString, String location). Similar to the other form of forceComponentGetUrl, this function allows the new URL to appear in a different page. Available location strings are:
  - "_self" Show in the window and frame that contain the applet.
  - "_parent" Show in the applet's parent frame. If the applet's frame has no parent

frame, acts the same as "_self".
    - "_top" Show in the top-level frame of the applet's window. If the applet's frame is the top-level frame, acts the same as "_self".
    - "_blank" Show in a new, unnamed top-level window.
    - Name Show in the frame or window named name. If a target named name does not already exist, a new top-level window with the specified name is created, and the document is shown there.
- public void setForeground (Color foreground). Sets the foreground color of the control.
- public void setBackground (Color background). Sets the background (fill) colour of the control.
- public void setForegroundBackground (Color foreground, Color background). This allows the programmer to set both foreground and background colours at once.
- public void setFont (String name, int style, int size). Sets the font of the control.
- public void displayMessageBox (String title, String caption). Displays a window with a caption and a single OK button.
- public int queryMessageBox (String title, String caption, int type). Allows the programmer to ask a question and get a user response.
- public void setCursor (Cursor cursor). Sets the cursor to a defined type.
- public FontCharacteristics getFontCharacteristics (...). These functions allow the server to query some of the FontMetric type information on the client. This includes the basic font size information and optionally the width of a string in a specified font.

Note: It is important to keep in mind that Amber controls are written to look and behave as the equivalent Java AWT controls. However Amber controls are not AWT controls and therefore are not guaranteed to implement all methods and properties of the standard AWT controls. As the Amber ComponentHandler's extend java.awt.Component there are a number of legacy functions contained in Component which are not supported in the Amber ComponentHandler. For this reason it is worthwhile when using Amber controls to check that the function being called is supported in Amber.

Panels

Panels are one of the more powerful and useful ComponentHandlers available in Amber. A panel is a container class which contains other ComponentHandlers. This allows powerful and varied application interfaces to be created which have the full functionality of a normal application.

Panels in Amber almost invariably extend the class `amber.server.panel.BasePanel`. This class contains substantial functionality making the use of panels in Amber simple and straightforward.

In its simplest form a panel can be considered as a drawing surface on which components are drawn. At the client browser panel components use different classes to those which reside

directly in the HTML. Typically there will be one to four RPanels defined in the HTML as applet tags. Each RPanel is capable of displaying any of the controls in the package `amber.client.panel`. The RPanel creates the panel controls dynamically as instructed by the server. It is possible to have panels (the panel class is `amber.client.panel.BasePanel`) created inside the parent RPanel, in this way a varied and rich interface can be created as required.

A panel is also capable of performing a simple set of drawing functions. We will describe each of these capabilities in turn.

Drawing Inside Panels

Panels support drawing operations as a series of commands to the panel which are executed in sequence. Later drawing operations may overlap earlier commands. Thus a piece of text could be laid over a filled rectangle. The functions relating to drawing are:

- setBorder. This allows the creation of a border around the entire panel. The border may be raised, lowered or none.
- addXXX. This adds an operation to the sequence of drawing operations. The possible operations are:
    - Draw Image.
    - Draw String.
    - Set Colour.
    - Draw Line.
    - Draw Rectangle.
    - Draw Oval.
    - Draw Arc.
- removeOperation. This command removes a specified drawing operation.
- removeAllOperations. This command removes all operations specified.

See the Appendix: Panel Specific Functionality for a more detailed description of the available functions.

Child ComponentHandlers in Panels

All panels can contain other controls which extend ComponentHandler (including other panels). Thus the user can define the controls which are displayed on a panel and create any user interface required.

All panels use a form of X/Y Layout where the location and size of the children controls are defined by describing their bounds within an X/Y grid. Using this schema the control boundaries are defined by setting the location of the top left corner of the control and the width and height of the control. No other layout type is possible for Amber panels. Functions exist in the `amber.server.panel.BasePanel` class which allow the manipulation of the controls inside the panel. These functions are:

- add. This function adds a ComponentHandler to the panel. The syntax of the command is:

  public void add (ComponentHandler newComponent, Object constraints)

  where newComponent is the ComponentHandler to add to the panel and constraints is an instance of amber.type.server.XYConstraints defining the size and location of the control.
  It is the act of adding the component to the panel which creates the visual control at the browser. For this reason if the component is not added to the panel it will not be possible to use it.
- remove. This function removes a control which was already added to the panel. This removes all the visual and messaging elements associated with the control. This is important to note when adding visual Frames (window controls, see below) to the panel. Only when remove is called is the window disposed of. The function syntax is:

  public void remove (ComponentHandler component) throws ComponentHandlerException

  where component is the ComponentHandler to remove from the panel.
- setBounds. This function redefines the X/Y/Width/Height of the control in the panel. Syntax:

  public void setBounds (ComponentHandler component, int x, int y,
  int width, int height) throws ComponentHandlerException

  public void setBounds (ComponentHandler component, Rectangle rect) throws ComponentHandlerException
- setLocation. A subset of setBounds, this function moves the component in the panel. The size of the control is unchanged. Function syntax is:

  public void setLocation (ComponentHandler component, int x, int y) throws ComponentHandlerException

  public void setLocation (ComponentHandler component, Point point) throws ComponentHandlerException
- setSize. The other subset of setBounds this function leaves the component location unchanged but resizes the control. Function syntax is:
  public void setSize (ComponentHandler component, Dimension dimension) throws ComponentHandlerException

  public void setSize (ComponentHandler component, int width, int height) throws ComponentHandlerException

See the Appendix: Panel Specific Functionality for a more detailed description of the available functions.

XYConstraints

This class is used when adding a component into a panel. It defines the bounds of the control and any other information required by the panel when creating the component at the remote browser. The information held in the class is:

- Component bounds (required). This is in the form X, Y, Width, Height coordinates which define the top left corner of the control and its physical size.
- Remote class name (optional). This string overrides the normal behaviour of the Amber panel. Normally the BasePanel queries the component to determine which control should be created at the remote browser. Should this value be set the class defined in the remote class name string is created instead. This is rarely required and should only be done with a full understanding of the requirements of Amber panels and their associated controls. One possible use of this would be the creation of a new remote panel control which responds to the same control set as another Amber component. Thus one ComponentHandler could be used to control two types of remote panel controls. A better way in this case would be to extend the original ComponentHandler and override the getPanelType function to return the correct remote class name.
- Parameters string (optional). This string can be used to define the initial state of the remote control. This is typically used to simplify the number of calls required to set up a remote control. For example a label usually only requires the text to be set. Rather than calling setText on the associated LabelHandler the label could be included in the parameters. The structure of the string varies depending on the component. The possible values of the parameters is defined in the documentation for the associated ComponentHandler.

Creating your own Panels

Almost all custom panels extend the class: amber.server.panel.BasePanel. This class is abstract. It is not possible to instantiate this class directly. For this reason the normal behaviour is to extend this class and add the required functionality. Should a simple panel be required the class amber.server.panel.GenericPanel should be used instead. The GenericPanel extends BasePanel to give a simple panel with only generic functionality.

When extending BasePanel the following abstract functions must be defined:

- defineComponents. This function is the basic set-up function for the panel class. It *must* be called from the constructor in the extended class. It typically defines the base structure of the components in the panel and the associated listeners. The initial filled state of the components is normally placed in fillControls which is called at the end of defineComponents. This structure can be relaxed if the panel will only ever be added once and never removed. This structure must be adhered to if the panel is to be a member of a panel template group (see below). Function syntax is:

  protected void defineComponents ( ) ;

- fillControls. This function is called to set up the initial state of the components in a

panel. It is called every time the panel is recreated in a panel template group (see below). For this reason functionality which should only be called once such as adding listeners should never be placed here unless each instance of the panel will only ever be added once to a container. Function syntax:

public void fillControls ( ) ;

- canClose. This function is used in panel template groups to determine if the panel can be replaced with another panel (see below). Return true if this is not required. Function syntax:

  public boolean canClose ( ) ;

- saveData. This function is called in a panel template group just before the panel is replaced (see below) to save the state information of the panel if required. Leave empty if this functionality is not required. Function syntax:

  public void saveData ( ) ;

As the panel must be connected to a corresponding visual element at the remote browser it is possible that the panel may not be connected initially. Typically during instantiation the panel will not be connected to the remote panel. Amber attempts to reduce issues of this type by queuing outgoing commands until they can be sent. Should code be required which must be guaranteed that the remote client is connected then this should be placed in the addNotify function. This function is called whenever the panel is set active (i.e. is connected to the remote component). The corresponding function for when the panel is made inactive is removeNotify.

Example BasePanel Code

The following is an example of a simple panel it has a label, an edit field and a button. When the button is pressed it compares the text against the string and displays a message box with the results.

```
package mypackage;

import java.net.* ;

import amber.type.server.* ;
import amber.server.exception.* ;
import amber.server.application.ApplicationHandler ;
import amber.server.panel.* ;
import amber.server.component.* ;
import java.io.Serializable;
import java.awt.event.*;

public class MyPanel extends BasePanel implements ActionListener
{
    private ButtonHandler button =
        new ButtonHandler ( getParentApplication() ) ;
```

```
private LabelHandler label =
    new LabelHandler ( getParentApplication() ) ;
private TextFieldHandler edit =
    new TextFieldHandler ( getParentApplication() ) ;

/**
    * This constructor is initialised by the parent application.
    * @param application. ApplicationInterface which is the parent
    * application.
*/
public MyPanel ( ApplicationInterface application )
{
    super ( application ) ;
    defineComponents ( ) ;
}

/**
    * This constructor is for HTML panels, in this case the ID
    * for this panel is also specified along with the parent
    * application.
    * @param id. int ID for this panel.
    * @param application. ApplicationInterface which is the parent
    * application.
*/
public MyPanel ( int id, ApplicationInterface application )
{
    super ( id, application ) ;
    defineComponents ( ) ;
}

/**
    * This function determines if the panel can be closed.
    * The derived panels must determine if this panel can close.
    * If this is not possible the function should return false.
    * @return boolean false if it is not possible to close this panel.
*/
public boolean canClose ( )
{
    return true ;
}

/**
    * This function is called to save any required information
    * in the panel.
    * This function is called externally when another panel
    * wishes to take
    * over the base panel or when closing the panel.
    * This function need not actually do something.
*/
public void saveData ( )
{
}

/**
    * This function should be called at the end of recreatePanel.
    * Its specific purpose is to set the controls to a known state once
    * they are created.
*/
public void fillControls ( )
{
    try
    {
        button.setLabel ( "Press Me" ) ;
```

```
        } catch ( Exception ex )
        {
            ex.printStackTrace ( ) ;
        }
    }

    /**
        * This function is called to define the components which are
        * a part of this panel.
        * This function is called by the constructor to set up
        * the normal static components and their location.
    */
    protected void defineComponents ( )
    {
        try
        {
            add ( label,
                new XYConstraints ( 40, 40, 300, 20,
                    "Enter the text and press the button" ) ) ;
            add ( edit, new XYConstraints ( 40, 65, 200, 25 ) ) ;
            add ( button,
                new XYConstraints ( 80, 100, 100, 20, "Press Me" ) ) ;
            // Add the button listeners
            button.addActionListener ( this ) ;

            // Now fill the controls with any values
            fillControls ( ) ;
        } catch ( Exception ex )
        {
            ex.printStackTrace ( ) ;
        }
    }


    /**
        * This function is called when the button is pressed.
        * @param e. ActionEvent containing information relating
        * to the event which occurred.
    */
    public void actionPerformed(ActionEvent e)
    {
        try
        {
            String value = edit.getText ( ) ;
            if ( value.equals ( "My Secret Text" ) )
            {
                displayMessageBox ( "Guess", "You guessed my secret!!!" ) ;
            } else
            {
                displayMessageBox ( "Guess", "Nope, try again" ) ;
            }
        } catch ( Exception ex )
        {
            ex.printStackTrace ( ) ;
        }
    }
}
```

Now let's look at the above example and discuss some of the more important details. First the class definition:

```
public class MyPanel extends BasePanel implements ActionListener
```

This means that the class is called MyPanel and extends the class BasePanel. This is normal for panels. It implements ActionListener which means it can respond to ActionEvents, this is how we will handle the button press.

Now lets look at the ComponentHandler declarations:

```
    private ButtonHandler button =
        new ButtonHandler ( getParentApplication() ) ;
    private LabelHandler label =
        new LabelHandler ( getParentApplication() ) ;
    private TextFieldHandler edit =
        new TextFieldHandler ( getParentApplication() ) ;
```

If we compare these declarations with those for the ApplicationHandler in the previous sections we can see that these declarations contain less arguments. Because these ComponentHandlers correspond to panel controls we do not have to specify the ID value for them. While it is possible to do so it is more convenient and safer to get the application to do so for us. The Amber button is ButtonHandler, the Amber label is LabelHandler and the Amber TextField (edit) control is the TextFieldHandler class.

Now lets look at the constructors:

```
    public MyPanel ( ApplicationInterface application )
    {
        super ( application ) ;
        defineComponents ( ) ;
    }

    public MyPanel ( int id, ApplicationInterface application )
    {
        super ( id, application ) ;
        defineComponents ( ) ;
    }
```

These constructors are two of many possible constructors for panels. In this simple case we assume that the panel will either be a sub panel of another panel (the top constructor) or will be attached to a panel defined by an HTML applet tag (the second constructor). As required by all Amber components the class is initialised by the parent application (which is required for messaging) and optionally the ID which will be used to connect to the correct remote client panel. Notice that the defineComponents function is called in the constructors. This allows the classes to define which sub-components will appear on the panel (see below). If this function is not called the panel will likely remain blank.

This natually leads to the defineComponents function:

```
    protected void defineComponents ( )
    {
        try
        {
            add ( label,
                new XYConstraints ( 40, 40, 300, 20,
```

```
                "Enter the text and press the button" ) ) ;
            add ( edit, new XYConstraints ( 40, 65, 200, 25 ) ) ;
            add ( button,
                new XYConstraints ( 80, 100, 100, 20 ) ) ;
            // Add the button listeners
            button.addActionListener ( this ) ;

            // Now fill the controls with any values
            fillControls ( ) ;
        } catch ( Exception ex )
        {
            ex.printStackTrace ( ) ;
        }
    }
```

The function basically consists of a series of add functions which add the various declared components to the panel. This instructs the remote panel to build the corresponding visual elements and adds the ComponentHandlers into the messaging system. The XYConstraints classes are used to define the characteristics of the controls including size, location and initial parameters text. The parameters text is passed unchanged to the remote component at the client browser. It is up to the remote component as to how this string is processed. In the case of the LabelHandler component the parameters text is the text of the label.

The action listener for the button is also added at this point. The final thing the function does is call the fillControls function to set up the initial state.

```
public void fillControls ( )
{
    try
    {
        button.setLabel ( "Press Me" ) ;
    } catch ( Exception ex )
    {
        ex.printStackTrace ( ) ;
    }
}
```

This function sets up the controls to a default state. In this example we set the label of the button to "Press Me". While it would have been possible to set the label in the XYConstraints parameters field for demonstration purposes we have done so in fillControls. The function fillControls is called every time the panel is recreated (see Panel Template Groups).

```
public boolean canClose ( )
{
    return true ;
}

public void saveData ( )
{
}
```

These functions are required for panels. They normally only have a use in panel template groups and hence in this example do little. For a more detailed explanation of these functions and their use see the Panel Template Group section below.

Finally let's look at the actionPerformed function which actually does the require functions of the panel

```
    public void actionPerformed(ActionEvent e)
    {
        try
        {
            String value = edit.getText ( ) ;
            if ( value.equals ( "My Secret Text" ) )
            {
                displayMessageBox ( "Guess", "You guessed my secret!!!" ) ;
            } else
            {
                displayMessageBox ( "Guess", "Nope, try again" ) ;
            }
        } catch ( Exception ex )
        {
            ex.printStackTrace ( ) ;
        }
    }
}
```

This function calls the function getText on the remote TextField to get the text the user entered. It then compares the value with the string "My Secret Text". If the user correctly entered this string then a message box window is displayed on the client with the title "Guess" and the caption "You guessed my secret!!!" otherwise a message box window is displayed with the caption "Nope, try again".

While this example is simple it highlights the requirements for creating custom panels and handling the events which the remote client components transmit.

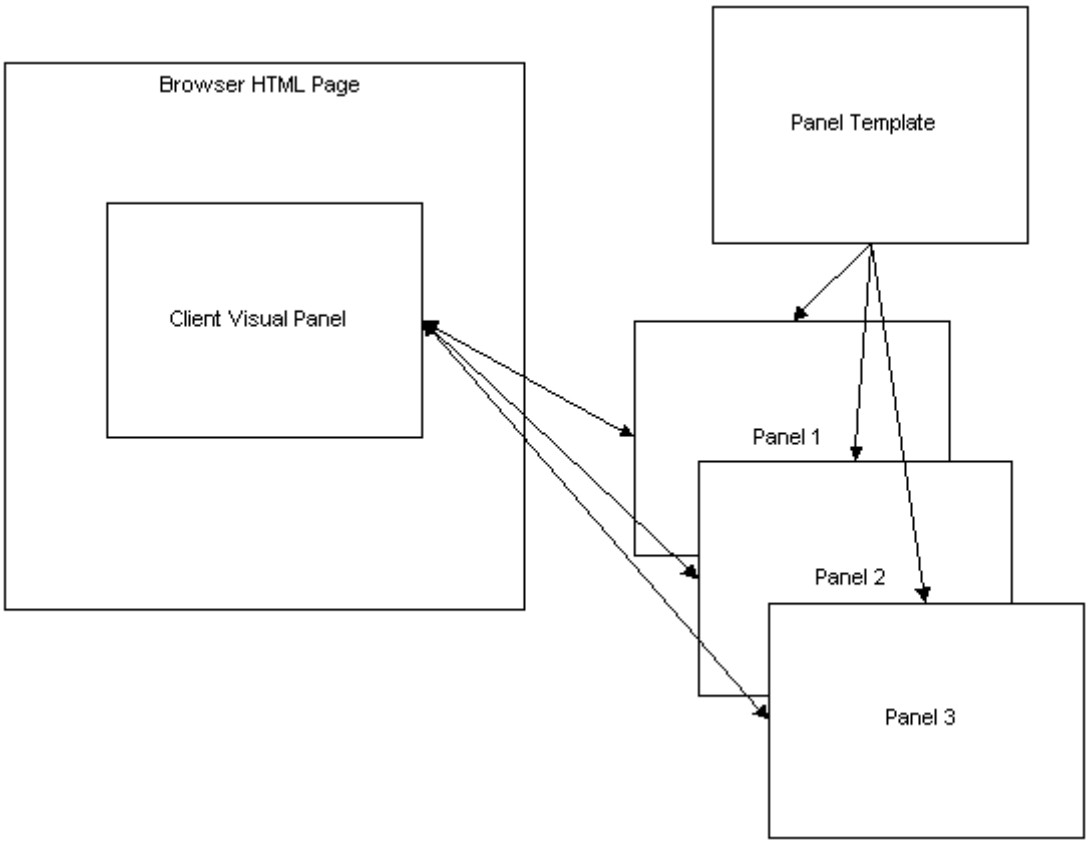
Panel Template Groups

A panel template group is a special situation where several panels *share* the same remote visual panel at the client browser. In this case the panels act as templates which are superimposed on the remote panel. This allows a multi-screen system where a browser panel can have different incarnations depending on requirements. This is useful in situations where the developer does not wish multiple frames (windows) but requires multiple screens of controls.

It is expected for a panel template group that a panel may be reconstructed more than once as the panels are swapped in and out of the client visual panel. The complexities of this operation are handled by the class amber.server.panel.PanelTemplateGroup

It is important to note that in a panel template group *all* panels share the same component ID. This is a unique situation in Amber as normally all controls have individual ID's.

The restrictions on panels in a panel template group are more stringent than for normal panels.



Again the panel extends amber.server.panel.BasePanel. All children must be added to the panel in the defineComponents function. While it is still possible to add controls in other parts of the code this must be done carefully. The BasePanel keeps a record of all ComponentHandlers added to it. When the panel is displayed in the client visual panel it attempts to recreate *all* the components it understands. Thus if a ComponentHandler is added later in the code, when the panel is recreated a second time this control will also be recreated. Should the later section of code also be executed again *two* versions of the control will appear. Thus code which dynamically adds ComponentHandlers must also ensure that old copies of the ComponentHandler are not there or are removed.

Only one panel can be selected (using the visual panel at once). This is done by using the extended constructor for BasePanel:

public BasePanel (ApplicationInterface pageHandler, PanelTemplateGroup panelGroup,

boolean selectedState) ;

where pageHandler is the ApplicationInterface (application) the panel is a part of, panelGroup is the PanelTemplateGroup the panel will be a part of and selectedState defines if the panel is to be the first displayed on the client visual panel.

When a panel is recreated on the client visual panel the following occurs:

- The function canClose is called on the previous panel. Should this panel return false a PanelHandlerException is thrown.
- Should the canClose function return true the function saveData is called on the previous panel. This allows the previous panel to save any data it may later require.
- The previous panel is then deselected. This stops the previous panel from responding to messages sent to the new panel.
- All the visual components on the client visual panel are removed.
- All the ComponentHandler's known to the BasePanel are recreated. Thus all the visual parts of the Amber ComponentHandlers are regenerated on the client visual panel in the last known position and size.
- The function fillControls is called. This allows the panel to be recreated and then the child controls set to a known predetermined state.

Example Panel Template code

In this section we will discuss the code needed to swap panels in and out of the client panel. For the purposes of this discussion we will assume that two panels: panel1 and panel2 share one visual panel (ID 0) defined in the HTML. The code is assumed to be part of the ApplicationHandler code.

The panels are instantiated using the following code:

```
private PanelTemplateGroup panels = new PanelTemplateGroup ( 0 ) ;

private MyPanel1 panel1 = new MyPanel1 ( this, panels, true ) ;
private MyPanel2 panel2 = new MyPanel2 ( this, panels, false ) ;
```

This code creates the panel template group and assigns a shared ID for the group of 0. It creates the two panels assigning them to the panel template group and making panel1 the first visual panel.

The following code changes the panel displayed on the client from panel1 to panel2.

```
try
{
        panels.setSelectedPanelHandlerAndCreate ( panel2 ) ;
} catch ( PanelHandlerException ex )
{
        System.err.println ( "Error changing panel" ) ;
```

```
        ex.printStackTrace ( ) ;
}
```

Frames

A Frame in Amber corresponds closely to a normal Java Frame. This is a stand-alone window with a title bar (caption). Frames may contain other components. Frames act almost identically to Panels. In fact amber.server.panel.BaseFrame extends the normal panel class amber.server.panel.BasePanel.

For the above reasons ComponentHandler's are added and removed in an identical fashion to BasePanel. BaseFrames have additional functionality required to handle the extended requirements inherent in stand-alone windows. This functionality is:

- BaseFrames can generate ComponentWindowEvents. In fact they are one of the few ComponentHandlers which do so. ComponentWindowEvents closely correspond to the standard Java WindowEvents. To listen to these events use the amber.awt.event.ComponentWindowListener interface (see Event Handling).
- get/setTitle. The title corresponds to the title bar caption. These functions allow the programmer to alter the caption of the window.
- setBounds. This function allows the top left location of the window and the window size to be set.
- setLocation. A subset of setBounds. This function changes the location of the window in the client screen.
- setSize. Again a subset of setBounds. This function changes the size of the window.
- toBack. Sends the window to the back of other windows. This function can be used to change the visual ordering of overlapping windows.
- toFront. Brings the window to the front of all other windows. Again this function changes the ordering of the windows on the client screen.

Frames are modeless windows. This means that the launching panel or frame continues to execute code and respond to events. To create a modal window (i.e. one which blocks all other input) use the ModalBaseFrame class (see below).

Modal Frames

A modal frame acts in a very similar manner to BaseFrame. It is relatively straightforward to create modal frames by extending BaseFrame and implementing the modal code directly. However, in general the normal method of creating a modal frame is to extend the class amber.server.panel.ModalBaseFrame. ModalBaseFrame extends BaseFrame adding in required functionality to make the BaseFrame modal.

Modal frames are handled slightly differently in Amber to the Dialog class in standard Java. A Dialog in Java requires a parent Frame when constructing the class. Amber does not have this restriction, *all* ComponentHandler classes have the capacity of becoming modal. In general, it

is not meaningful for normal controls such as buttons etc to trap all incoming events.

For a ComponentHandler to become modal the class must call the function setComponentModal with a boolean argument having a value of true. From this point on the ComponentHandler gains *all* events sent from the client. To terminate modality call setComponentModal with a false argument. The function setComponentModal is protected and must be called by an extending class.
ModalBaseFrame contains the additional function: `show`. Calling `show` causes the modal frame to set the Frame visible and block all execution and events to the parent class which called it. Thus the parent code will stop at the show call until the window internally calls setComponentModal (false). For this reason the ModalBaseFrame extended class *must* call setComponentModal (false) (typically as the result of a button press or window event) or the application will not continue execution.

Example ModalBaseFrame

The following is a simple example of a modal frame which contains a single button which closes the window.

```
package mypackage;

import amber.server.application.ApplicationInterface ;
import amber.server.panel.* ;
import amber.type.server.* ;

public class MyModalFrame extends ModalBaseFrame implements ActionListener
{
    private ButtonHandler button =
        new ButtonHandler ( getParentApplication() ) ;

    public MyModalFrame ( ApplicationInterface pageHandler )
    {
        super ( pageHandler ) ;
        defineComponents ( ) ;
    }

    public void fillControls ( )
    {
        button.setLabel ( "Press Me" ) ;
    }

    protected void defineComponents ( )
    {
        add ( button, new XYConstraints ( 10, 10, 100, 25 ) ) ;
        button.addActionListener ( this ) ;
        fillControls ( ) ;
    }

    public void actionPerformed ( ActionEvent e )
    {
        setComponentModal ( false ) ;
    }
}
```

The code that would display this modal frame is as follows:

```
private void showFrame ( )
{
      // Create the frame class
      MyModalFrame myFrame = new MyModalFrame (
            getParentApplication() ) ;
      // Tell the panel to create this frame at the remote client
      // This displays the frame at location 10, 10
      // with a size 300*500 pixels
      add ( myFrame, new XYConstraints ( 10, 10, 300, 500 ) ) ;
      // Display the frame and block until it returns
      myFrame.show ( ) ;
      // Remove the frame
      remove ( myFrame ) ;
}
```

Unlike standard Java Frames it is the parent panel which creates the visual Frame on the client. For this reason it should be added to the panel. Calling remove on the panel disposes of all the associated visual resources for the frame. The XYConstraints class defines the starting location of the top left point and the physical size of the frame window itself.

Active Properties

There are several properties which are not valid unless the application is connected to the client. When the ApplicationHandler is first instantiated the application is not connected to the server and any commands are left in a pending state. It is important not to put code in the constructors which depend on parameters that have not been set when the application is instantiated. These parameters include:

- Client Screen Size. This is attained using the getClientScreenSize function in the ApplicationHandler. It returns the Dimension of the screen at the client.
- Page ID. This is the page identifier which was used to identify which ApplicationHandler extended class to instantiate.
- Page Sub ID. Similar to Page ID it can be used to identify the ApplicationHandler or it may be used to pass optional configuration information to the application.
- Log object. While the primary Core log object is valid, the log object contained in the ApplicationHandler will not be.
- The core socket over which the application communicates with the client components.

These properties can be guaranteed valid in the following places:

- ApplicationHandler. The parameters are correctly set by the time the start (Socket newConnection) function is called. Override this function and place your own code after the super.start (newConnection) call, i.e.

```
public void start ( Socket newConnection )
{
      // Initialise the ApplicationHandler
      super.start ( newConnection ) ;
```

```
        // Perform your own initialisation here
}
```

- ComponentHandler extended classes. This includes BasePanel and BaseFrame. When the ComponentHandler is activated (i.e. connected to the remote client) the function addNotify() is called. When the ComponentHandler is deactivated (i.e. disconnected from the client) the function removeNotify() is called. By overriding these functions it is possible to create code which depends on the active parameters.

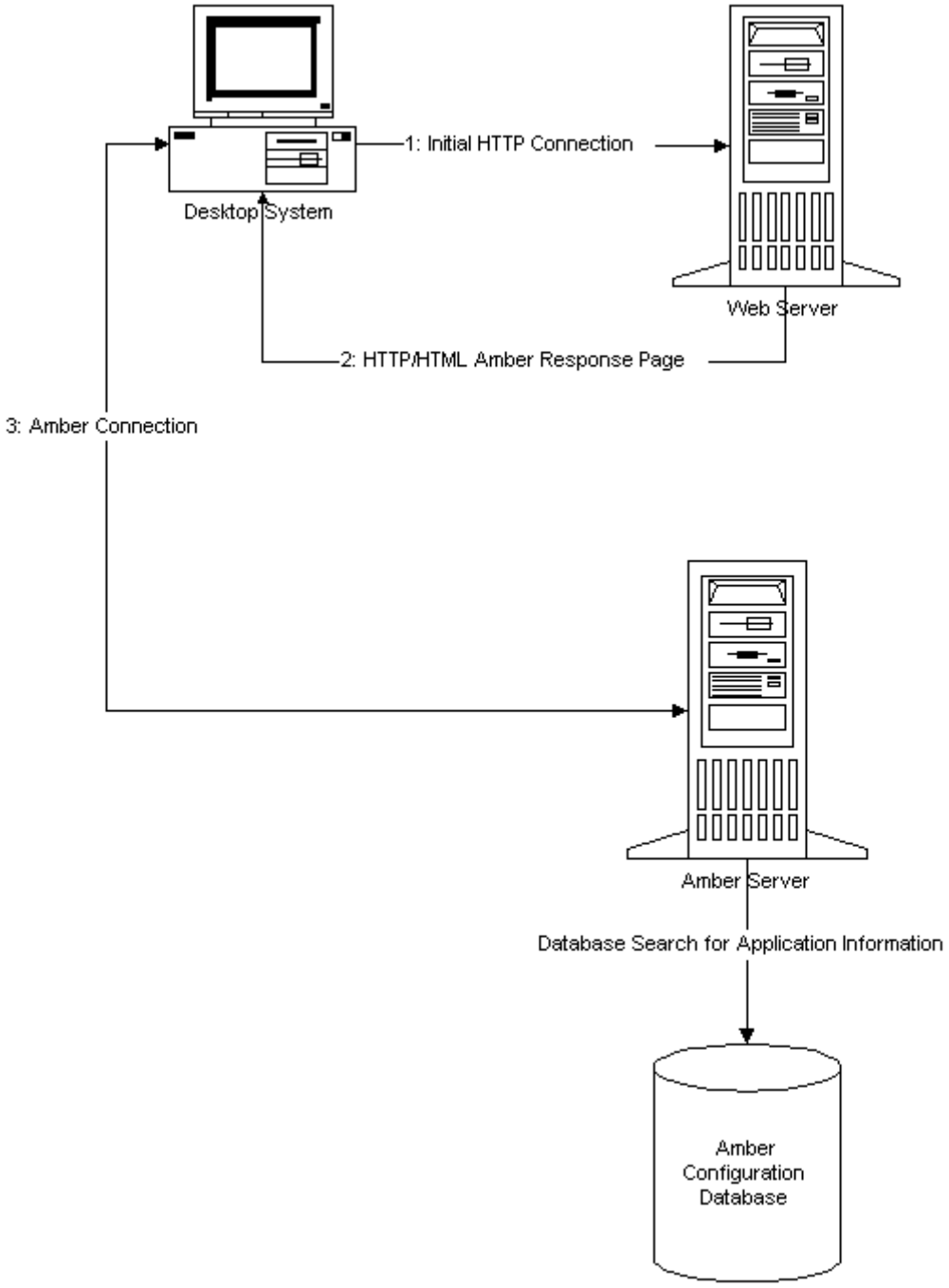
```
public void addNotify ( )
{
        // Call the super addNotify
        super.addNotify() ;
        // Perform your own initialisation here
}
```

Running Amber Applications

Introduction

There is a strong correlation between the HTML page and an Amber application. It is the HTML page which is served by the web server to the client browser. The applets (HTML resident components) initialise and connect to the Amber server typically installed on the same system as the Web Server. The Amber server then selects the correct application (ApplicationHandler extended class) and runs it, initialising it with the connection to the remote client. Thus the normal initialisation sequence for an Amber application is as follows:

1. The client browser connects to the Web Server and requests an HTML page containing Amber controls.
2. The browser loads the Amber client jar containing the Amber client classes.
3. The Amber client (controls) applets initialise and the ID 0 control establishes a connection to the specified Amber server, typically listening on port 21384.
4. The Amber server reads the connection packet and extracts the Page ID and Page Sub ID fields.
5. The Amber server reads the Amber configuration database looking for information corresponding to the input Page ID and Page Sub ID.
6. If these values are located in the configuration database the corresponding ApplicationHandler extended class is instantiated.
7. The ApplicationHandler is initialised with required information. This includes the Page ID, Page Sub ID and the socket connection to the remote ID 0 control.
8. The ApplicationHandler is started and manages the connection from this point.



Configuring the Amber Server

As can be seen the Amber server must be configured to recognise the incoming connection request and correctly start the right application class. Crucial to this process is the Amber Configuration Database.

To create and run an Amber application the following process must be followed.

1. The base Amber application must be created. This is a class which extends the class amber.server.application.ApplicationHandler. The application may contain panels/frames or other more simple Amber controls. Thus although an Amber application starts with an ApplicationHandler class it usually contains a substantial number of additional classes to handle the processing requirements for the application.
2. The HTML page corresponding to the application is created. This is standard HTML which is formatted in the normal way but contains one or more Amber specific controls. The configuration for the ID 0 control also contains connection information such as:
   a. Page ID.
   b. Page Sub ID (optional).
   c. Amber Server IP address.
   d. Amber Server IP Port number (optional).
3. The HTML page is added to the Web Server at the required location. Amber makes no restrictions as to HTML page location. For the default installation the Web root location is: amberserver/live/htdocs.
4. The application classes are copied into a location within the Amber class path. The typical location is either:
   a. amberserver/classes.
   b. amberserver/usr/classes.
   Both these locations are in the normal class path for the Amber server.
5. The required database configuration for the application is added to the Amber Server configuration database. This allows the Amber server to match the incoming Page ID/Page Sub ID information against the application class.
6. The Amber server is restarted if necessary. If this is the first time that the classes have been added this is not necessary. If the Amber server *has* loaded these classes before the internal class loader will have cached the application classes. In this case the server must be restarted.
7. Connect the browser to the HTML page and determine if the application loads properly.

It is very important when debugging applications to keep looking at the AmberServer.log file. This file, typically located in the amberserver directory is where the Amber server displays errors. More information is logged than displayed on the Java console.

If a page is not loaded the console will contain the error:

```
Page 2004, -1 not found or could not be loaded, please check the
AmberServer.log
```

This error will occur if the application could not be started. This may occur if the Page ID/Page Sub ID cannot be found *and if the application threw an exception when instantiated and initialised*. The exception thrown will be logged in the AmberServer.log and can be used to debug the application initialisation process.

Page ID/Page Sub ID

The Page ID and Page Sub ID's are used to identify which ApplicationHandler class is to be started when a connection is received at the Amber server. For this reason they must be unique. Furthermore the entry in the HTML page must match the corresponding entry in the configuration database. It is possible to configure the database configuration entry to ignore the Page Sub ID when identifying the correct ApplicationHandler. This allows the Page Sub ID to be used to carry simple integer information between the Web Server and the Amber Server. More extensive information should be stored in a database accessible by both the Web and Amber servers.

Database Configuration

The Amber configuration database contains two tables needed by the Amber server (in fact it is possible to have only one table if only one of normal/secure connections are required). The table names are configurable in the AmberServer.properties file but default to "Pages" and "SecurePages". Both tables have the same format and field names. There are example script files for some databases in the amberserver/database/scripts directory. These script files contain both the table creation SQL along with the row update SQL code for the demonstration Amber applications. The structure of the tables is as follows:

Name	Type	Description
PageID	integer (32 bit)	The identifying page number used to locate the application
PageSubID	integer (32 bit)	A secondary number used to refine the location process. This is optional (see RequireSubID field).
Handler	varchar (255)	The fully qualified name of the class to load. This must extend ApplicationHandler. The ".class" should *not* be appended to the end of the class name. The package should be included in the class name.
Info	varchar (255)	A human readable string giving information on the application.

Status	integer	Can be one of 3 states: • 0. Pending, waiting for connection. • 1. Active, the application is running. • 2. Ended, the program has terminated. These states are not guaranteed to correctly reflect the state of the application. Should be set to 0.
CreationDate	date	The date the application was added to the database.
ExpirationTime	integer (64 bit)	How long the application will run before being terminated. This is specified in milliseconds. A value of -1 indicates no expiration.
RequireSubId	integer (boolean)	Should the Amber server use both the PageID and PageSubID fields when identifying the correct application. If set to false the PageSubID field is ignored. A 1 is considered true else false.
Transient	integer (boolean)	If this flag is set to true (1) the database row is deleted once a valid connection is detected. This allows one-shot applications to be created.
Multiple	integer (boolean)	If this flag is true (1) only one copy of an application can run at any one time. Multiple connections requests after the initial connection will be rejected until the current application is terminated.
PublicHandler	integer (boolean)	When true (1) this application is available to everyone who requests it. This is more applicable for systems which can request available applications. Browser based connections cannot access this information.
MailConnection	integer (boolean)	If this is set to a 1 (true) and the server is capable of connecting to an SMTP server, the Amber server will e-mail a specified location every time a connection is received. The following fields must also be correctly set up.
MailMessage	varchar (255)	The body of the e-mail message to send.
MailRecipient	varchar (255)	The person to send the e-mail message to.
MailSmtpServer	varchar (255)	The SMTP server to attempt to transmit the e-mail message through.

MailSender	varchar (255)	The sender of the e-mail message.

At the current point in time the following fields are unimplemented:

- ExpirationTime.
- Multiple.

A standard creation SQL query to create the database table could look like:

```
CREATE TABLE Pages (
  PageID integer NOT NULL,
  PageSubID integer NOT NULL,
  Handler varchar(255) NOT NULL,
  Info varchar(255),
  Status integer NOT NULL,
  CreationDate date,
  ExpirationTime integer NOT NULL,
  RequireSubID integer NOT NULL,
  Transient integer NOT NULL,
  Multiple integer NOT NULL,
  PublicHandler integer NOT NULL,
  MailConnection integer NOT NULL,
  MailMessage varchar(255),
  MailRecipient varchar(255),
  MailSmtpServer varchar(255),
  MailSender varchar(255),
  PRIMARY KEY (PageID,PageSubID)
);
```

This is a simple version of the query. More complex versions with default values are more common however this displays the basic table structure. An example SQL query to add an application to the database would be:

```
INSERT INTO Pages VALUES (10000,-1,'demo.DemoPage1','Simple Amber
Demonstration',0,curdate(),-1,0,0,1,1,0,NULL,NULL,NULL,NULL);
```

This query adds the application demo.DemoPage1 to the database. It will be loaded for any connection with a Page ID of 10000. The connection is permanent allowing multiple connections with no expiration time. No mail message will be sent on connection.

Text File Application Configuration

It is possible for simple applications to avoid the use of the configuration database and replace it with a text file. This text file is specified in the config/amberserver/AmberServer.properties file in the:

AmberServer.NormalPreCacheConnectionFile
AmberServer.SecurePreCacheConnectionFile

properties. This points to a file which contains the configuration in a text form. For this reason for simple requirements this may be more convenient. The format of the file is:

PageId, PageSubId, PageSubIdRequired, ApplicationHandler class, Information String, Status, Creation Date, Expiration Time, Start Time, Transient, Multiple Connections, Public, Mail Connection, Mail Recipient, Mail Sender, Mail SMTP Server, Mail Message

with each field separated by commas. These fields have the same meanings as the corresponding database fields. An example of this file can be found in the config/amberserver/ConnectionPrecache.dat file.

Server Architecture

The Amber server acts as a flexible framework which can be extended by adding specialised server objects. The general structure of the server is shown in the following diagram.

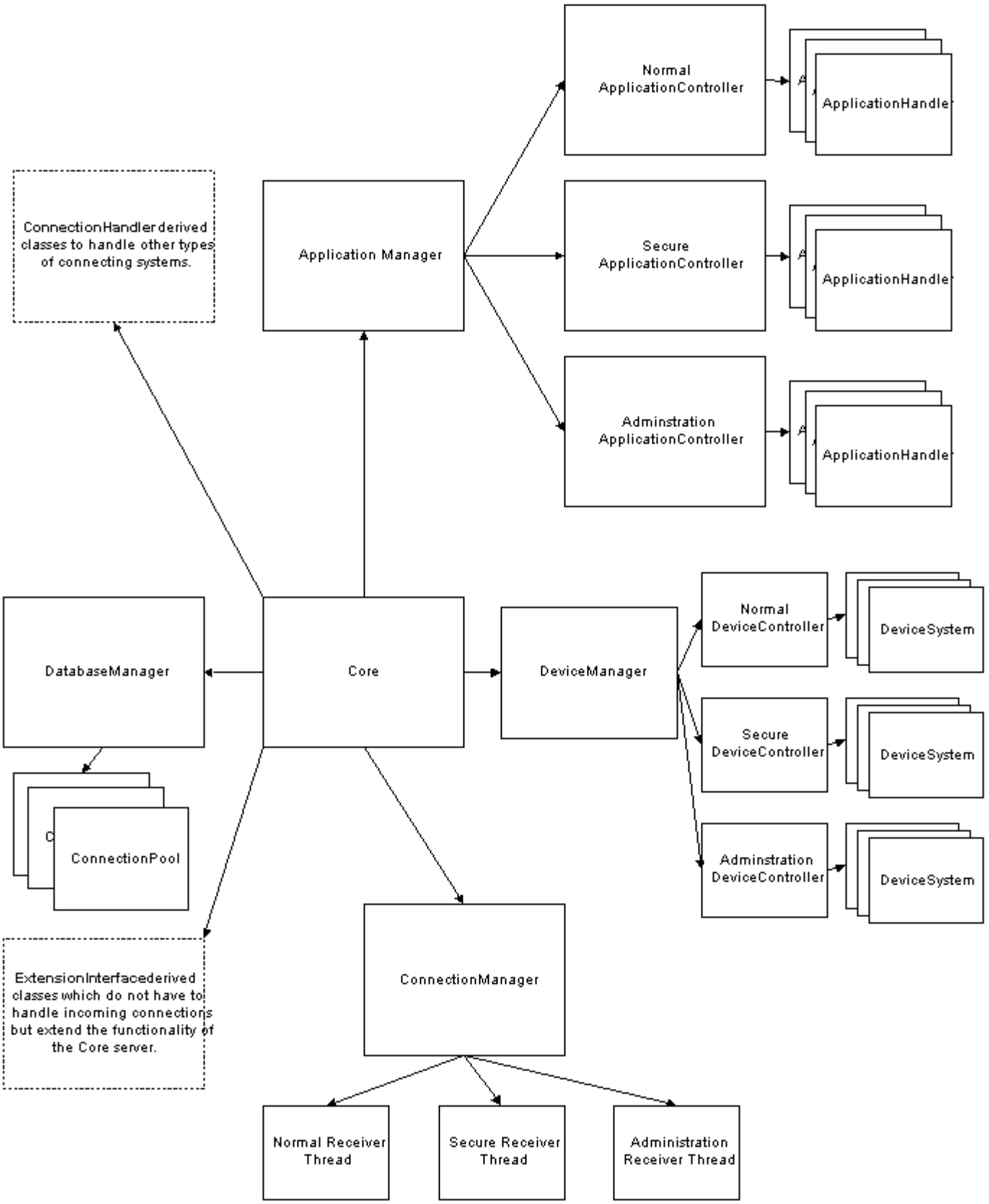
The centre of the server system is the amber.server.manager.Core class. It is this class which connects all the various server components of the AmberServer together. The core system contains a variety of extension modules which provide the required additional functionality needed to make the Amber server operate. These extension modules are:

- Connection Manager. The responsibility for this module is to handle incoming connections. The connections are processed for type and passed to the correct module to be handled in the appropriate manner.
- Application Manager. This module takes the incoming connections and associates the corresponding ApplicationHandler derived class which will from this point take control of the connection and handle events and transmit commands to the remote client.
- Database Manager. This module handles multiple databases and creates pools of connections which are available for use by the various modules. This mechanism is also available for use by the various ApplicationHandler objects.
- Device Manager. This is responsible for managing a distributed device architecture which allows devices on distributed systems on the network to be available for use by applications and server modules.

It is relatively straightforward for additional modules to be added to the server core. Amber supports two different types of extension modules. These are:

- Other Connection Managers. The current ConnectionManagers available are Browser based connections, Java application based connections and Device connections. It is possible to extend this to include a new type of incoming connection. These new managers would implement the ConnectionHandler interface. The Application Manager and Device Manager are examples of this type of module.
- General extension modules. These expand the functionality of the primary server core but do not directly interact with the client connections and the associated applications. Timer or RMI interface modules would be examples of this type of module.

The additional modules are added to the server by adding the required information into the AmberServer.properties file in config/amberserver. For more information on this process see the Amber Installation Guide.



Amber Server Core

The primary engine of the Amber server is the amber.server.manager.Core class. This class contains all other elements which make up the Amber server. It is always possible to gain access to the Core object. Access is via the static function getCore() i.e.

```
public static Core getCore ( )
```

This function returns the handle to the global Core object.

Once access to the Core has been gained the programmer has access to functionality allowing the user to access the various managers and also:

- getAmberRoot. This function returns a string containing the path to the Amber server root directory at the server.
- getDocumentRoot. Part of the functionality required by Amber is Web based. Images loaded by image controls, HTML files loaded from within Amber etc are located relative to the Web server document root. This function returns a string pointing to the Web server document root. This allows Amber applications to programmatically generate web content.
- getProperties. This returns the PropertyHandler object which controls the AmberServer properties file (config/amberserver/AmberServer.properties). This allows properties to be set in the server properties file and accessed from an application.
- setLoggingLevel. This changes the logging level allowing the amount of information logged by the server to be altered.

See the Appendix: Amber Server Core Functionality for a more detailed description of the available functions.

Using Databases

One of the more common requirements for a server application is the manipulation of database information. Amber supports multiple databases and allows multiple connections to the databases to be created and handled in a simple and consistent manner.

The Amber database interface utilises the standard Java Data Base Connectivity (JDBC) standard giving the widest possible flexibility of use for programmers. This allows the programmer a simple unified mechanism for database access. It becomes the responsibility of the Amber server to maintain the database connections freeing the programmer to concentrate on program logic rather than the requirements of establishing and maintaining database connections.

Database Manager

The Amber database interface consists of a single Database Manger which controls a collection of ConnectionPool's (`amber.server.manager.database.ConnectionPool`). Each ConnectionPool is responsible for managing database access to a single database. The ConnectionPool contains a series of Connection's (`java.sql.Connection`) to the database.

The Database Manager has the following utility functions available for use:

- createNewConnectionPool. This function allows the programmer to create a new ConnectionPool object dynamically. The normal method for adding new ConnectionPools is to add the database configuration details to the config/database/DatabaseManager.properties file. However this function allows the dynamic creation of a required ConnectionPool.
- getAllConnectionPools. This function returns a Vector of all possible ConnectionPools understood by the Database Manager.
- getConnectionPool. This function gets a specific ConnectionPool defined by name. All ConnectionPools should have unique names. Should more than one ConnectionPool have the same name the behaviour of the system is undefined.
- getTotalPools. This returns the number of ConnectionPools registered with the Database Manager.

The primary Database Manager object can be accessed from the Core object in Amber server.

Connection Pools

Under normal circumstances the programmer will operate most often with one or more ConnectionPool objects. A connection pool is an interface to a database. The programmer requests a connection to the database from the connection pool and uses this connection to perform database transactions. When finished the programmer releases the connection back to the pool allowing other applications to use it.

A connection pool is configured to have a minimum and maximum number of connections. The minimum number of connections are opened to the database when the connection pool is created. If all the minimum number of connections are in use and another connection is requested a new connection is created for the requester. This will continue until the maximum number of connections is exceeded at which point a requester will be given a null connection.

Connection pools are identified by name. When created the connection pool must be allocated a unique name or the wrong connection pool may be returned from the Database Manager. Each Connection Pool has a unique log file in which database errors are logged.

Operations possible on ConnectionPool objects are:

- getAvailableConnections. Returns the number of available connections for this connection pool. This is the total number of connections minus the number allocated.
- getConnection. This returns a java.sql.Connection object to the database. This allows any normal JDBC operation to be performed against the database. The Connection is checked out to the requesting application and is unavailable for other use until released.
- getPoolName. This is the name of the ConnectionPool. This should be a unique value.
- getMaximumConnections. Returns the absolute maximum connections which can be connected to the target database.
- getMinimumConnections. Returns the minimum number of connections which are connected to the target database when the ConnectionPool is created.
- getTotalConnections. This returns the total number of connections which have been currently created for connection to the database. This should be compared with getMinimumConnections and getMaximumConnections. This value may change as more connections are created under load.
- releaseConnection. This function takes an allocated Connection and releases it back into the pool of available connections for use by other applications.

It is important to note that a connection should only be requested just before use and then released. There is no connection penalty to getting a connection as all pooled connections are constantly kept connected to the database. Maintaining a permanent connection to the database limits the number of simultaneous sessions the Amber server can support to the number of available database connections.

Example Database Code

The following code demonstrates a simple code segment which accesses a database. Normally the application would get a handle to the DatabaseManager and ConnectionPool once rather than each time the function is called but the example demonstrates the principles involved.

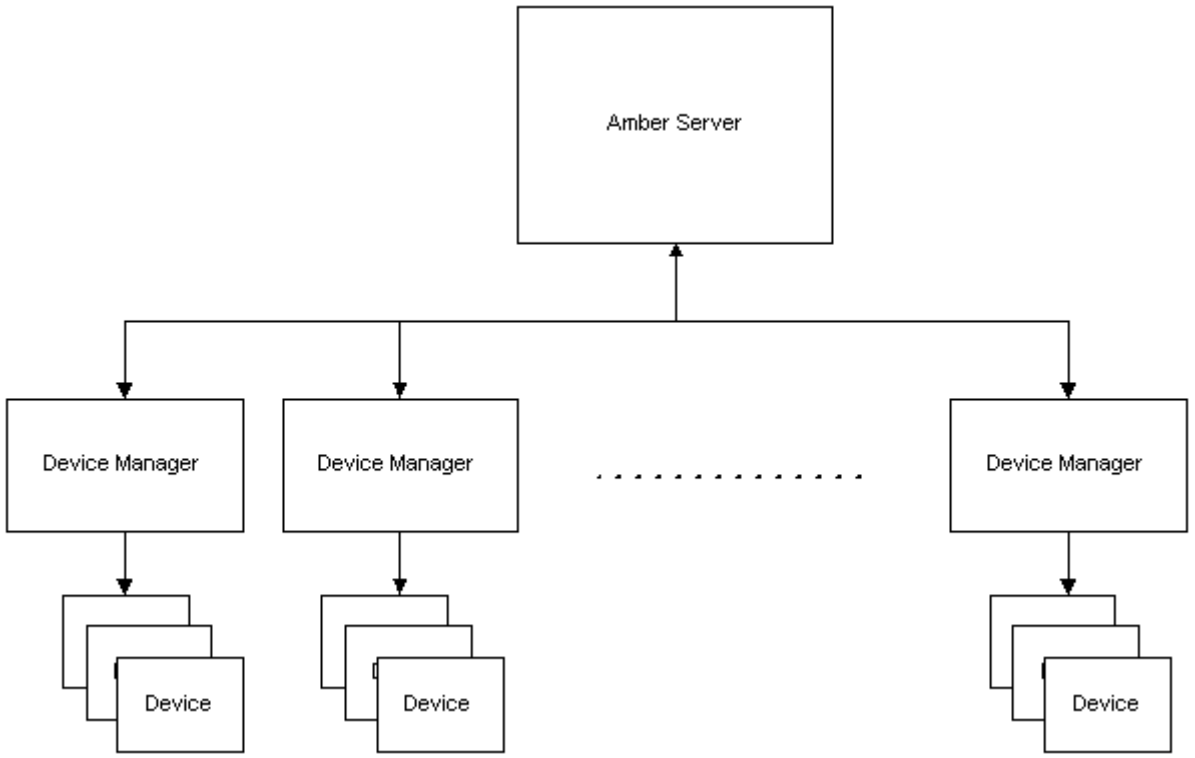
```
import amber.server.manager.* ;
import amber.server.manager.database.* ;
import java.sql.* ;


...


public int getRowCount ( )
{
    Core core = Core.getCore ( ) ;
    DatabaseManager dm = core.getDatabaseManager ( ) ;
    ConnectionPool cp = dm.getConnectionPool ( "TestDB" ) ;
    ResultSet rs = null ;
    Statement statement = null ;
    Connection connection = null ;
    int value = 0 ;
    try
    {
        connection = cp.getConnection ( ) ;
        statement = connection.createStatement ( ) ;
        rs = statement.executeQuery ( "Select count(*) from infotab" ) ;
        value = rs.getInt ( 1 ) ;
    } catch ( Exception ex )
    {
        ex.printStackTrace ( ) ;
    } finally
    {
        if ( rs != null )
            rs.close ( ) ;
        if ( statement != null )
            statement.close ( ) ;
        cp.releaseConnection ( connection ) ;
    }
    return value ;
}
```

Using Devices

Introduction

The Amber server system is capable of handling a distributed device system which is flexible and powerful, allowing the best use of available distributed resources. These devices can be such things as modems, printers, bar-code readers along with the more simple serial and parallel ports. Devices are assumed to be connected to a number of nodes on a distributed network. Each node is capable of managing the physical requirements of one or more devices and promoting the capabilities of these to the main server system. So, this makes it possible to have printers connected to different computers and have an Amber application print to them irrespective of where they are located. Clients running remote browsers across the world can connect to Amber servers and control the devices. Thus full remote computer management becomes simple and easy.

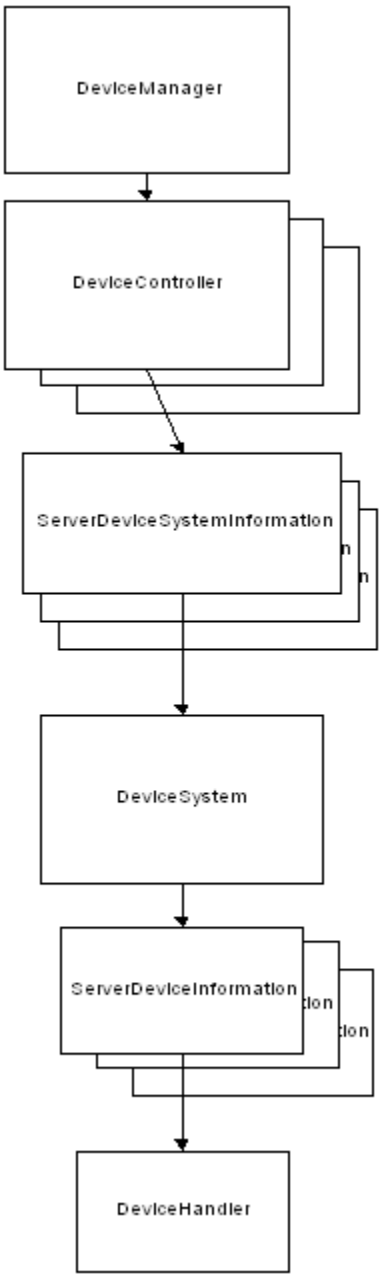
When each Device Manager initialises, it connects and registers it's characteristics and resources with the Amber Server. The Amber Server then creates a server side entity to control the remote Device Manager and adds the remote devices to its device pool. These devices are then available for use by applications running on the Amber Server.



Using the Amber Device System

The Amber Server allows applications to query the device manager looking for a particular type of device. The device manager also contains an elementary security system allowing devices to reside inside layers of security. When the application requests a device it also provides a security level. This level is used by the device manager to identify devices which are allowed to the application. Higher security levels allow more secure devices to be accessed.

The structure of the device manager system is as follows:



Selecting Specific Devices

To access the device manager inside Amber the following code could be used:

```
Core core = Core.getCore ( ) ;
DeviceManager dm = core.getDeviceManager ( ) ;
```

This manager can then be queried for available devices. This is done using the following functions:

- getDeviceSystems. This function returns a Vector of the class amber.type.server.ServerDeviceSystemInformation which contains information about the available DeviceSystems. The DeviceSystem classes control the DeviceManagers on the client systems. The syntax of the function is:

  public Vector getDeviceSystems (int type)

  The type integer defines the type of connection which connects the remote DeviceManager to the Amber server. This constant corresponds to that defined in the amber.server.manager.connection.ListenerThread. Currently these values are:
  - NormalConnection (normal connections).
  - SecureConnection (encrypted connections).
  - AdministrationConnection (non-encrypted administration related connections).

- getMatchingDevices. This function is the normal mechanism for querying available devices. It requests the DeviceManager for any devices which match the input type and sub-type values. These values can be found in the Device Type Values appendix. The syntax of the function is:

  public Vector getMatchingDevices (int connectionType, int securityLevel, int deviceType, int deviceSubType)

  where the arguments are:
  - connectionType. This corresponds to the type of connection (see above).
  - securityLevel. This is the security level used to define which devices the application is allowed to see. The numbers range from 0 upward.
  - deviceType. int defining the major type of device to be searched for. A -1 means any device.
  - deviceSubType. int defining the minor type of the device to be searched for. A -1 means any device defined by the deviceType field.

  The function returns a Vector of class amber.type.server.DeviceMatch which contains information on the device and the corresponding DeviceSystem.

Example of Requesting a Device

The code to request a serial device could look as follows:

```
import amber.server.manager.connection.* ;
import amber.server.manager.device.* ;
import amber.type.server.* ;


...
    Core core = Core.getCore ( ) ;
    DeviceManager dm = core.getDeviceManager ( ) ;

    Vector devices = dm.getMatchingDevices (
            ListenerThread.NormalConnection, 0, 1, 1 ) ;
    for ( int j=0; j<devices.size(); j++ )
    {
        DeviceMatch currentDevice = (DeviceMatch)devices.elementAt ( j ) ;
        String systemName = currentDevice.getDeviceSystemName ( ) ;
        String deviceName = currentDevice.getDeviceName ( ) ;
            ...
    }
```

Once the device has been selected it must be allocated. To allocate a device call the accessDevice function for the corresponding DeviceSystem. A device can be opened for exclusive or shared access. Some devices can only be exclusively allocated. Thus the code to use a device detected in the previous code could look like:

```
    DeviceMatch currentDevice = devices.elementAt ( 0 ) ;
    if ( !currentDevice.getDeviceSystem().accessDevice (
        currentDevice.getDevice(), DeviceSystem.ExclusiveAccess ) )
    {
        displayMessageBox ( "Device Error",
            "Unable to gain exclusive access to the device" ) ;
    } else
    {
        // Do something with the device
    }

    // Release the device now that we are finished
    if ( !currentDevice.getDeviceSystem().releaseDevice (
        currentDevice.getDevice() ) )
    {
        displayMessageBox ( "Device Error",
            "Unable to release exclusive access to the device" ) ;
    }
```

Using the Device Handler

Controlling a device in the application is by interacting with the corresponding DeviceHandler class. The device handlers extend the class `amber.device.server.DeviceHandler`. These classes have a similar function to the ComponentHandler classes for normal visual components.

The DeviceHandler is accessed from the DeviceMatch object by calling the function getDevice(). Once access to the device has been requested and received the remote device can be manipulated by calling functions in the corresponding DeviceHandler. There are a variety of functions common to all devices however each device contains extra functionality as required by the type of device.

Functions common to most DeviceHandler's are:

- public void close ( ). This function closes the device. All outstanding requests are flushed and terminated.
- public DriverInfo getDriverInfo ( ). This returns some simple information on the driver. This includes a version number and a descriptive string.
- public int getStatus ( ). This returns an integer status of the device.
- public boolean open ( ). This opens the device for use. This should not be confused with the accessDevice() function in the DeviceSystem which merely tells the server to allocate the device to this application.
- public byte [] read ( ). Reads all the available input information from the device.
- public void reset ( ). Resets the device back to a known default state.

Example of Using a Device Handler

This example shows how a panel in an application could write data to a serial device. It assumes that the device was allocated using similar code to the previous example. The example opens the device when the open button is pressed, closed when the close button is pressed and the text from a text field is transmitted via the serial port when the send button is pressed. The required declaration code for the controls is also assumed.

```
private TextFieldHandler txtIn = new TextFieldHandler (
    getParentApplication() ) ;
private ButtonHandler btnOpen = new ButtonHandler (
    getParentApplication() ) ;
private ButtonHandler btnClose = new ButtonHandler (
    getParentApplication() ) ;
private ButtonHandler btnSend = new ButtonHandler (
    getParentApplication() ) ;

private SerialHandler serialPort ;
...

public void actionPerformed ( ActionEvent e )
{
    try
    {
        Object source = e.getSource ( ) ;
```

```
            if ( source == btnOpen )
            {
                // Open the device
                if ( !serialPort.open ( ) )
               {
                    displayMessageBox ( "Open Error",
                         "Failed to open serial port" ) ;
               }
            } else if ( source == btnClose )
           {
                // Close the device
                serialPort.close ( ) ;
            } else if ( source == btnSend )
            {
                 // Get the data from the text field
                String data = txtIn.getText() ;
                // Send it to the serial port,
                // assume default coding for the string
                serialPort.write ( data.getBytes() ) ;
            }
        } catch ( Exception ex )
        {
            ex.printStackTrace ( ) ;
        }
    }
```

Creating Components

Introduction

One of the most complex operations in Amber is to create a new component for use in the system. This is owing to the deeper knowledge of how Amber works that is required to create a component. Before a component can be used by Amber the following must be done:

1. The visual element of the component must be created. This is typically one Java class which extends Canvas or Panel and performs some visual function. Examples of this would be ticker components etc.
2. A client wrapper class must be created. This performs the messaging which is required to receive server commands and send client events.
3. A server component handler must be created. This is the server side component which handles the messaging requirements from the server to the client wrapper class.

Once this is done the client parts are added to the deployment jar and the server classes added to the Java classpath. This will then allow the component to be used in any applications required. We will now treat each of the three parts of the creation process in turn.

Creating the Visual Element

The visual element is used both as a mechanism for displaying information to the user and gathering user input events from the user. In general, Amber visual elements are created in exactly the same way as a normal Java visual bean would be created. For this reason, it is outside the scope of this document to describe how a visual element is created. The following recommendations however should be observed:

- Ensure that the visual element does just what it should do and no more. The visual element is a moderate proportion of the information downloaded to the client browser. For this reason any code not strictly required should be removed. In general, functions which are never to be used by the client wrapper class should not be written. Remember, the larger the download size, the longer the user must wait before they can use the application.
- Put the majority of the more complex processing back on the server by placing it in the server component handler. Again this relates to the download size. This will be a balancing act between the download size and the messaging overhead. As a rough rule of thumb continuous messaging between client and server should be avoided, however action type events should be processed at the server.
- Make sure that the visual element operates correctly in Java version 1.1. The majority of browsers using this visual element will only have Java 1.1 resident on the browser. This typically eliminates the use of Swing components for visual elements.

- Ensure that the visual element does not trap events needed by Amber, these include: Key, Focus, Mouse and Mouse Motion Events. If you wish Focus events to work correctly ensure you override isFocusTraversable from Component to return true.
- Minimise the number of new support classes needed by the visual element. There is an overhead for every new class created. This can increase the download substantially. The optimal number of visual element classes to be downloaded is either 0 or 1.
- Ensure that the visual element ultimately extend the java.awt.Component class. While this is less of a requirement for HTML resident components it is vitally important for panel resident components.

The best way to develop the visual element is to create a fully functional, bug free visual element in an applet or application first. This can then be used by the client wrapper classes.

Creating the Client Wrapper Class

There are two possible wrapper classes which correspond to how the client component is used. The difference between the two depends on whether the component will be placed directly into the HTML page or as a component residing on a panel.

Before the component is written the commands it will respond to and the events it transmits must be known. In general, the component will send the following events:
- Component Events.
- Key Events.
- Focus Events.
- Mouse Events.
- Mouse Motion Events.

These events are automatically handled by an Amber Component. Amber does not generate or handle Component Events.

When selecting the commands they must be chosen in such a way that the class size is kept to a minimum. This class, along with the visual element, is deployed to the client browser. For this reason the class size should be kept to a minimum.

Commands for a component are byte values allocated starting at the offset:

```
RConstants.ExtensionCommandBaseId
```

Packets sent from the server are allocated command IDs and so too are any response packets sent back to the server. Thus a command which returns a value back to the server such as getSelectedIndex would use two command IDs one for the command and the second for the response.

It is often useful to classify commands into groups of like commands which share a common command ID, with the first data parameter being an integer defining which group command this is. It is however unlikely that the number of commands defined for a client component

will overrun the allowed possible command values.

Having decided on the commands/responses and the events the component will handle we now move onto the details of creating client wrappers. We will now treat each of the two types of client wrapper classes in turn:

HTML Resident Components

These components extend the amber.client.RComponent class. The RComponent class in turn extends a standard applet. It is the applet which resides in the HTML code. Normally the requirement is that the visual element will occupy the entire drawing surface of the applet. For this reason the normal situation is to use a BorderLayout for the applet and add the visual element using BorderLayout.CENTER. The RComponent handles most of the requirements involved in messaging to the Amber Server. It handles the events generated by the visual component and performs initial parsing and processing of incoming Amber AATP packets.

An example of a simple Amber HTML component is:

```
package mypackage;

import java.awt.*;
import java.applet.* ;
import java.awt.event.*;
import java.io.* ;

import amber.type.Packet ;
import amber.server.component.* ;

/**
  * Extension to RComponent that presents the user with a new control.
  *
  * @see amber.client.RComponent
*/
public class MyHtmlComponent extends RComponent
{
    private BorderLayout borderLayout = new BorderLayout();
    private MyControl control = new MyControl();

    public void init()
    {
        // Set up the base RComponent
        super.init() ;

        // Set up the layout for this component
        this.setLayout(borderLayout);
        this.add ( control, BorderLayout.CENTER ) ;

        // Add in additional listeners in here

        // Set up the default listeners
        addBaseListeners ( control ) ;

        // Process any component specific applet parameters here

    }
```

```
    public void processPacket(Packet packet)
    {
        super.processPacket(packet) ;

        try
        {
            switch (packet.getCommand())
            {
                case MyHandler.FirstCommand:
                    {
                        // Put the relevent command processing code here
                        // The dataDecoder variable contains the
                        // command parameters
                        int myFirstParameter = dataDecoder.readInt ( ) ;
                    }
                    break ;

                case MyHandler.SecondCommand:
                    {
                        // Reset the output data stream
                        encoderBytes.reset ( ) ;
                        Packet p = new Packet(
                            MyHandlerler.SecondCommandResponse
                            Packet.ServerId, getTrueId(), false) ;
                        dataEncoder.writeInt ( control.getValue() ) ;
                        p.setData ( encoderBytes.toByteArray ( ) ) ;
                        forwardPacket(p) ;
                    }
                    break ;
            }
        }
        catch (Exception err)
        {
            err.printStackTrace() ;
        }
    }

}
```

Let us now look at the various part of the control wrapper code. First is the declaration section:

```
    private BorderLayout borderLayout = new BorderLayout();
    private MyControl control = new MyControl();
```

In this case the first declared variable is the BorderLayout required to push the control to the full size of the HTML applet. The other variable is the visual element control you created in stage 1 of the component generation process.

The next part of the code is the init function. This overrides the init in the RComponent and ultimately the applet. The code appears as follows:

```
    public void init()
    {
        // Set up the base RComponent
        super.init() ;

        // Set up the layout for this component
```

```
        this.setLayout(borderLayout);
        this.add ( control, BorderLayout.CENTER ) ;

        // Add in additional listeners in here

        // Set up the default listeners
        addBaseListeners ( control ) ;

        // Process any component specific applet parameters here

    }
```

This function does the following:

- It calls the super.init function to initialise the RComponent code.
- It sets up the BorderLayout as the layout manager for the applet.
- It adds the visual control into the applet.
- It sets up any extension event listeners beyond the base set automatically handled by RComponent. This is typically calling control.addXXXListener (this) to make this component a listener for any events handled the visual control.
- The addBaseListeners (control) function is called to set up the default set of listeners to listen to the visual control.

Lastly is the processPacket function. This function handles the incoming commands to the client coomponent. It is composed of the following parts.

- It calls the super.processPacket function to handle the default commands which any Amber control responds to. These include setVisible, setEnabled, displayMessageBox etc.
- This example control handles two commands. Each of the commands highlights different aspects of handling incoming packets. It should be noted that the command constants are typically stored in the ComponentHandler rather than the client component. This typically reduces the download size.
    - `MyHandler.FirstCommand`. This command shows how a parameter is read from the incoming packet. While it is possible to process the packet data directly, when the packet is received the RComponent processes the packet and sets up an amber.type.AmberInputStream (extending java.io.DataInputStream) called dataDecoder with the packet data. This allows the client code to read this information simply by using the stream read functions.
    - `MyHandler.SecondCommand`. This command shows how a response packet is assembled and transmitted back to the server. An amber.type.AmberOutputStream (extending java.io.DataOutputStream) dataEncoder is always available for use. The reset function is called to remove any previous data. A packet is created for transmission. The information to be sent is streamed to the output stream and the encoded byte array is attached to the packet using the setData function. The packet is then transmitted using the forwardPacket function.

Panel Resident Components

A panel resident component is similar in function to an HTML resident component however it

is designed to only function inside an Amber panel or frame. This object contains additional functionality to allow the addition or removal of the component from the panel. The HTML and panel resident components are independent of each other. It is possible to create one and not the other depending on requirements. Both client components do however require a server side ComponentHandler to connect to.

A panel resident component will reside in a panel or frame which uses the layout manager amber.awt.XYLayout. This layout allows the component visual element to be placed in an absolute location and size within the confines of the panel. It is possible for the component to be at any size or location on the panel and this should be taken into account when designing the visual element.

Let us now look at an example panel resident component:

```
package mypackage ;

import amber.type.Packet ;
import amber.server.component.* ;

import java.io.* ;
import java.awt.* ;

import java.util.StringTokenizer ;

/**
  * Panel Resident Component.
  * Uses the same ComponentHandler as the corresponding remote component.
  *
  * @see amber.client.panel.BaseControl
*/
public class MyPanelComponent extends BaseControl
{
    /**
        * Initialize the component.
    */
    public MyPanelComponent ( )
    {
        super ( ) ;
    }

    /**
        * This is the main initialisation function for this class.
        * @param mainParent. RContainer handle to the panel this is a
        * component of.
        * @param data. int [6] containing information useful to the
        * component in the order ID, eventMask, x, y, w, h.
        * @param parameters. String containing any required parameters
        * separated by '|'
    */
    public void init ( RContainer mainParent, int [] data,
        String parameters )
    {
        visualObject = new MyControl ( ) ;
        super.init ( mainParent, data, parameters ) ;
    }

    /**
        * Process the parameter values.
```

```
    * @param parameter String containing component specific parameters.
*/
public void parseParameters ( String parameter )
{
    if ( parameter == null )
        return ;

    // The parameters is just the string
    // Decode the string into the various required parameters
}

/**
    * Add in the listener handlers to the input object.
    * @param listener The listener which will receive the event
    * messages.
*/
public void addListeners ( BaseComponent listener )
{
    super.addListeners ( listener ) ;
    // At this location add in the listeners
    // For this example we will add in an action listener
    ((MyControl)visualObject).removeActionListener(this) ;
    if ( eventEnabled [ RConstants.ActionEventIndex ] )
        ((MyControl)visualObject).addActionListener(this) ;
}

/**
    * This routine contains the logic to parse a packet and perform
    * actions based on the packet's contents.
*/
public void processPacket(Packet packet)
{
    super.processPacket(packet) ;

    try
    {
        switch (packet.getCommand())
        {
            case MyHandler.FirstCommand:
                {
                    // Put the relevent command processing code here
                    // The dataDecoder variable contains the
                    // command parameters
                    int myFirstParameter = dataDecoder.readInt ( ) ;
                }
                break ;

            case MyHandler.SecondCommand:
                {
                    // Reset the output data stream
                    encoderBytes.reset ( ) ;
                    Packet p = new Packet(
                        MyHandlerler.SecondCommandResponse
                        Packet.ServerId, getTrueId(), false) ;
                    dataEncoder.writeInt (
                       ((MyControl)visualObject).getValue() ) ;
                    p.setData ( encoderBytes.toByteArray ( ) ) ;
                    forwardPacket(p) ;
                }
                break ;
        }
    }
```

```
        catch (Exception err)
        {
            err.printStackTrace() ;
        }
    }
}
```

As can be seen the panel resident component wrapper has substantial similarities to the HTML resident component. Let us look at each section in turn.

In this case there is no global declaration area. The global panel visual element handle is stored in a global variable visualObject which is declared in BaseControl. This is a handle to a java.awt.Component class so it is important that the visual element class ultimately extend this class. Also note that no LayoutManager is defined. This is handled for you by the panel and therefore no additional coding is required.

This class contains a default constructor (i.e. a constructor with no parameters) this is required by the Amber system. The class calls the super constructor to initialise the Amber specific requirements.

In a similar way to the HTML resident component, most of the complex initialisation is performed in the init function:

```
    public void init ( RContainer mainParent, int [] data,
        String parameters )
    {
        visualObject = new MyControl ( ) ;
        super.init ( mainParent, data, parameters ) ;
    }
```

This function instantiates the visual element and then calls the super initialisation function to finalise the initialisation. The parameters passed to the function include the container in which this component is added, an int array containing the parameters used when creating the object. This includes the ID given to the component, the event mask used for messaging (which events are sent to the server), and the parameters string which corresponds to the parameters string in the XYConstraints class.

As part of the initialisation process two other functions are called. These are parseParameters and addListeners. These functions allow the component to initialise specific parts of the panel resident component.

```
    public void parseParameters ( String parameter )
    {
        if ( parameter == null )
            return ;

        // The parameters is just the string
        // Decode the string into the various required parameters
    }

    public void addListeners ( BaseComponent listener )
    {
        super.addListeners ( listener ) ;
        // At this location add in the listeners
        // For this example we will add in an action listener
```

```
        ((MyControl)visualObject).removeActionListener(this) ;
        if ( eventEnabled [ RConstants.ActionEventIndex ] )
            ((MyControl)visualObject).addActionListener(this) ;
    }
```

The parseParameters function allows the initial parameters to be set in the component. The string specified in the XYConstraints function is passed to the component on construction and appears in the parseParameters function. This string can be of any format required however the normal characteristics of the string is of a series of parameters separated by '|' characters.

The addListeners function is called on initialisation and also whenever the event mask is altered under server control. For this reason it is more complicated than that of the HTML resident control. Old listeners must be removed and then depending on the eventEnabled array which is internal to BaseControl and updated before the addListeners function is called add the required extension listeners to the visual element. Standard listeners are added for you by calling the super.addListeners function. The argument to the function is the ultimate component responsible for transmitting the events however you will almost never use this parameter.

Finally comes the processPacket function. As can be seen this function is almost identical to the same function in the HTML resident component. The primary difference relates to using the visualObject handle and coercing it to the correct type.

Creating the Server ComponentHandler

This is the final stage of creating an Amber component. This class handles the server side requirements of connecting to the client control. This is a complex operation however most of the complex part of the connection process is hidden from sight in ComponentHandler and other related classes.

To create the new component handler for the created client component the class must extend the class amber.server.component.ComponentHandler. It is this class which will handle the messaging and connection requirements for us.

To continue with our previous example the following code would connect to the client component:

```
package myserverpackage;

import amber.type.* ;
import amber.client.* ;
import amber.server.exception.* ;
import amber.server.application.ApplicationInterface ;
import java.awt.event.*;
import java.util.*;
import java.io.Serializable ;

/**
  * This class handles the requirements for the manipulation of the packets
```

```
  * moving to and from my component.
  *
  * @see amber.server.application.ApplicationInterface
  * @see amber.server.component.ComponentHandler
  * @see amber.type.server.XYConstraints
*/
public class MyComponentHandler extends ComponentHandler implements
Serializable
{
    public static final byte
        FirstCommand = RConstants.ExtensionCommandBaseId + 0,
        SecondCommand = RConstants.ExtensionCommandBaseId + 1,
        SecondCommandResponse = RConstants.ExtensionCommandBaseId + 2 ;

  /**
        * Default constructor.
    */
    public MyComponentHandler()
    {
        this ( InvalidId, null ) ;
    }

    /**
      * The initialising constructor. This constructor dynamically
      * requests a valid ID value from the ApplicationInterface.
      * @param pageHandler The handle to the main ApplicationInterface
      * which handles the functions of page overall.
    */
    public MyComponentHandler ( ApplicationInterface pageHandler )
    {
        this ( pageHandler ) ;
  }

    /**
        * The initialising constructor.
        * @param id The int containing the id of the corresponding remote
        * component residing on the browser.
        * @param pageHandler The handle to the main ApplicationInterface
        * which handles the functions of page overall.
    */
    public MyComponentHandler ( int id,
          ApplicationInterface pageHandler )
    {
        super ( id, pageHandler ) ;
    }

    /**
        * This function handles the incoming packets from the component
        * residing on the browser. This code may generate
        * events if there is a requirement for the ApplicationInterface to
        * handle the packet.
        * @param packet Packet containing the packet to handle.
        * @exception amber.server.exception.ComponentHandlerException
        * containing the error information.
    */
    protected void handlePacket ( Packet packet )
        throws ComponentHandlerException
    {
        switch ( packet.getCommand ( ) )
        {
            case SecondCommandResponse:
```

```
                // Do special processing here if required
                break ;
            default: // No idea get the super class to handle it
                super.handlePacket ( packet ) ;
                break ;
            }
    }

    /**
        * Call the first command.
        * @param item String containing the component information.
        * @exception amber.server.exception.ComponentHandlerException
        * contains any errors performing this command.
    */
    public void setFirst ( String item ) throws ComponentHandlerException
    {
        // Encode the data
        // Reset the output data stream
        try
        {
            encoderBytes.reset ( ) ;
            dataEncoder.writeLongUTF ( item ) ;
            byte [] data = encoderBytes.toByteArray ( ) ;
            // Create the packet and send it
            Packet command = new Packet ( (byte)FirstCommand, (short)id,
                Packet.ServerId, data.length, data, false ) ;
            sendPacket ( command ) ;
        } catch ( Exception ex )
        {
            throw new ComponentHandlerException (
                "setFirst error: " + item, ex ) ;
        }
    }

    /**
        * Returns the selected index
        * @return int containing the selected index.
        * @exception amber.server.exception.ComponentHandlerException
        * contains any errors performing this command.
    */
    public synchronized int getSelectedIndex()
        throws ComponentHandlerException
    {
        int retValue = -1 ;
        try
        {
            // C2reate the packet and send it
            Packet command = new Packet ( (byte)SecondCommand, (short)id,
                Packet.ServerId, false ) ;
            // Block and wait for the reply, maximum delay 20s
            PacketData currentPacket = sendPacketAndBlock ( command,
SecondCommandResponse ) ;
            if ( currentPacket != null )
            {
                    // Convert the data to an int
                    retValue = currentPacket.getDataDecoder().readInt ( ) ;
            } else
            {
                  throw new MissingResponseException ( "getSelectedIndex:
failed to get response" );
            }
        } catch ( Exception ex )
```

```
        {
            throw new ComponentHandlerException (
                "getSelectedIndex error", ex ) ;
        }
        return retValue ;
    }

    /**
        * This function returns the panel component to use with this class.
        * @return String containing the panel type to use.
    */
    public String getPanelType ( )
    {
        return "mypackage.MyPanelComponent" ;
    }
}
```

This class is broken into several sections. The first is the declaration of the command byte values. Remember these are the command values which are used in the packet to identify which commands the component will respond to. These must start at the value: RConstants.ExtensionCommandBaseId which defines the starting value for user commands. For each command to the client component there is a corresponding command byte value. In the case of commands which return a value there will be two one for the command to the client and one for the response from the client.

Following the declaration section are the constructors for the object. These differ slightly however the general functionality is the same. The constructors instantiate the internal global objects (in this case there are none). Constructors which allow the system to allocate an ID number (i.e. where the ID is not passed in the constructor) cannot be used for HTML resident components as the ID is predefined.

After the constructors is the handlePacket function. This has a similar function to the processPacket function in the client components. It is designed to handle packets from the client specific to this component. Standard events are handled automatically and should not be handled here.

```
    protected void handlePacket ( Packet packet )
        throws ComponentHandlerException
    {
        switch ( packet.getCommand ( ) )
        {
            case SecondCommandResponse:
                // Do special processing here if required
                break ;
            default: // No idea get the super class to handle it
                super.handlePacket ( packet ) ;
                break ;
            }
    }
```

This function checks the packet to identify what the command is. For this example there is only one type of incoming packet we are interested in, the response packet to our SecondCommand. In the case of a normal response packet there is no special processing

required. In general, for normal components this function need not be created. It is normally only required where a new type of event is to be handled from the client. If the packet is not this response the code passes the packet onto the super class to process. Should the super class be unable to identify the command it throws an amber.server.exception.UnknownCommandException.

The next two functions present the command interface to the server applications. It presents two functions which correspond to our command packets. The first function is a command which does not expect a response, setFirst:

```
    public void setFirst ( String item ) throws ComponentHandlerException
    {
        // Encode the data
        // Reset the output data stream
        try
        {
            encoderBytes.reset ( ) ;
            dataEncoder.writeLongUTF ( item ) ;
            byte [] data = encoderBytes.toByteArray ( ) ;
            // Create the packet and send it
            Packet command = new Packet ( (byte)FirstCommand, (short)id,
                Packet.ServerId, data.length, data, false ) ;
            sendPacket ( command ) ;
        } catch ( Exception ex )
        {
            throw new ComponentHandlerException (
                "setFirst error: " + item, ex ) ;
        }
    }
```

This function resets the encoder, writes the string into the dataEncoder which is an AmberOutputStream which encodes the string for transmission. The encoded byte array is then extracted from the encoder. The packet is created using our command value and the encoded data. Finally the sendPacket function is called to send the packet to the remote client.

The second function (getSelectedIndex) is more complex. It is similar to setFirst however the latter part of the function handles the client response. The code is as follows:

```
    public synchronized int getSelectedIndex()
        throws ComponentHandlerException
    {
        int retValue = -1 ;
        try
        {
            // Create the packet and send it
            Packet command = new Packet ( (byte)SecondCommand, (short)id,
                Packet.ServerId, false ) ;
            // Block and wait for the reply, maximum delay 20s
            PacketData currentPacket = sendPacketAndBlock ( command,
SecondCommandResponse ) ;
            if ( currentPacket != null )
            {
                        // Convert the data to an int
                retValue = currentPacket.getDataDecoder().readInt ( ) ;
            } else
            {
```

```
                throw new MissingResponseException ( "getSelectedIndex:
failed to get response" );
            }
        } catch ( Exception ex )
        {
            throw new ComponentHandlerException (
                "getSelectedIndex error", ex ) ;
        }
        return retValue ;
    }
```

The function initialises the return value to a default value. The command is sent in a similar manner to before however the function called to send the packet is: sendPacketAndBlock. This takes two arguments: the packet to send and the expected response packet ID. This is the ID which will be looked for when the response is detected from the client. This function will not return until the response packet is received or the timeout value is exceeded. The return from this function is the response packet or null if no response is received. This PacketData class contains the return response packet and the AmberInputStream which corresponds to the data from the packet. Thus by calling the getDataDecoder() function in the PacketData, the data held in the packet can be read.

The last part of the ComponentHandler code is a utility function used internally by Amber to match the ComponentHandler to the corresponding panel resident component. It is ignored by HTML resident components.

```
    public String getPanelType ( )
    {
        return "mypackage.MyPanelComponent" ;
    }
```

This function returns the class which must be instantiated at the remote client when this ComponentHandler is added to a BasePanel. It can be overridden by specifying a different remote class name in the XYConstraints class.

Non Visual Components

It is possible for panel resident components to be *non-visual*. A non-visual object acts in a similar way to a visual object however it is not added to a panel. To create a non-visual object is simple, just use nonVisualObject rather than visualObject to hold the object. The component would exist and respond to messages but would not be added to the panel.

The function canAdd is used for components such as windows which are Component extended objects but which should not be added to the client panel. The default response for this function is true which tells the panel to add the visualObject to the panel. By overriding this function to return false the visualObject will not be added to the panel.

Appendix: Common Amber Applet Tags

The following applet parameter tags are common to all Amber controls:

- Component. This is the client class which is the display object. In this case the display object is a panel (amber.client.panel.BasePanel) however any of the normal components in amber.client.panel can be used.
- Coordinates. This is most often used by the PFrame class however it defines a set of 4 numbers which will be passed to the panel component. These correspond to the 4 numbers which are passed in the XYConstraints class. They are comma delimited with no spaces. If this is not specified the numbers 0, 0, -1, -1 are used.
- ConnectionModule. Optional. Used to specify an alternative mechanism for connecting to the Amber server (see below).
- EventMask. Optional. Used to define which events the Amber client will transmit to the server. This is used to reduce the network traffic sent over the network to that which is absolutely required. This parameter is rarely used as the server can set this information also. The following are the values of the event mask and the corresponding events they enable. Multiple events are OR'ed together to produce the final event mask.
  - ActionEventMask = 1,
  - TextEventMask = 2,
  - KeyTypedEventMask = 4,
  - KeyPressedEventMask = 8,
  - KeyReleasedEventMask = 16,
  - MouseClickedEventMask = 32,
  - MousePressedEventMask = 64,
  - MouseReleasedEventMask = 128,
  - MouseEnteredEventMask = 256,
  - MouseExitedEventMask = 512,
  - ItemEventMask = 1024,
  - WindowEventMask = 2048,
  - MouseDraggedEventMask = 4096,
  - MouseMovedEventMask = 8192,
  - FocusLostEventMask = 16384,
  - FocusGainedEventMask = 32768 ;
- ExtensionX. Where X is a monotonically increasing integer starting at 0 (i.e. Extension0). Optional. This is a series of properties which are to be passed back to the Amber Server by the the ID 0 component. These extension properties can be read from the ApplicationHandler using the getRemoteProperties() function. The value of the Extension param is in the form "name|value" where name is the name of the property and value is the value given to the property.
- ID. This is the ID number of the Amber control. This ID number is used to link the

client control with the server ComponentHandler. These numbers must be unique and there must always be a control with an ID of 0. The ID 0 control is required to have several other parameters which relate to its role as master control.

- PageId. Required, only used by ID 0. This integer is used by the Amber server to identify the type of application to be attached to this client.
- PageSubId. May be optional, only used by ID 0. This integer is used to further refine the type of application to attach. The server can be configured to ignore this integer when identifying the correct application. In this case the number may be ignored or can be used to pass additional information to the starting ApplicationHandler at the server.
- Port. Optional, only used by ID 0. This is the port to use when connecting to the Amber server. The default port if one is not specified is 21384.
- Secure. Optional. Default value is 0. When set to 1 this indicates that the connection will be encrypted. Implementation is specific to the connection module used (see below).
- Server. Optional, only used by ID 0. This defines the location of the server to which the Amber client system will connect. This may be either an IP (Internet Protocol) address or a standard dotted notation address such as `amber.clearfield.com`. If no address is specified then the base address of the server in the HTML URL is used.
- ServerSHA. Optional. Used for secure socket connections. This is the SHA-1 of the RSA encryption key of the server. This is used to guard against man in the middle attacks for the Amber secure connections. This value is generated using the amber.net.HashRSAPublicKeyFile program.

Connection Modules

Amber supports an extensible system of connection modules which allow a variety of protocols to be used connecting the client to the server. At this point there are two available types of module. They are:

- Socket. This opens a direct socket connection to the Amber server. This is the most efficient and has the lowest overhead in terms of data transfer. This system requires no additional support from the web server. The disadvantage of this system is that it requires that the socket connection on the Amber ports (21384-21386) be able to be made.
- HTTP. This utilises the HTTP protocol to transfer the AATP packets. This has a higher data transfer requirement and data latency, however it will work on any normal HTTP connection allowing Amber to operate even through proxy servers. It requires servlet support at the Web Server for the server side of the connection system.

Client

To use the connection module at the client specify the connection module to use by the ConnectionModule parameter in the ID 0 applet. Multiple modules can be specified and the system will iterate through the specified modules in order attempting to make a connection to the Amber Server. The ConnectionModule parameter specifies the main class of the

connection module. This is a class which implements the interface amber.client.ConnectionModule.

Here is a relatively complete example:

```
<param name ="ID" value ="0">
<param name ="ConnectionModule0" value ="amber.client.SocketConnection">
<param name ="ConnectionModule1" value ="amber.client.HttpConnection">
<param name ="CONNECTIONURL" value ="/servlet/AmberReceiverConnect">
<param name ="SENDURL" value ="/servlet/AmberReceiverReceive">
<param name ="RECEIVEURL" value ="/servlet/AmberReceiverSend">
<param name ="SERVER" value ="amber.clearfield.com">
<param NAME = "Port" VALUE = "21384">
<param NAME = "PAGEID" VALUE = "2000">
```

As can be seen there are two connection modules available for this ID 0 component. The first module attempted will be the amber.client.SocketConnection module which attempts to use a direct socket connection to connect to the Amber Server. The second module amber.client.HttpConnection will attempt an HTTP connection to the Amber Server should the socket connection fail. Each module uses different parameters to completely configure it. We will now discuss each in turn.

Socket Connection Module

The properties used by the SocketConnection Module are:

- ConnectionModuleX. Where X is the order in which this module is used starting at 0. Optional. This is the default module should no connection module be specified. This parameter is required if more than one connection module is used. The value of this parameter is "`amber.client.SocketConnection`".
- Port. Optional. This is the socket port number at the Amber Server used to attempt a connection. The default is 21384.
- Secure. Optional. Default value is 0. When set to 1 this indicates that the connection will be encrypted. This option requires the additional security classes in amber.net.
- Server. Optional. This specifies the IP address of the server to connect to. The default is the HTML URL base server address.
- ServerSHA. Optional. Used for secure socket connections. This is the SHA-1 of the RSA encryption key of the server. This is used to guard against man in the middle attacks for the Amber secure connections. This value is generated using the `amber.net.HashRSAPublicKeyFile` program.

HTTP Connection Module (Enterprise edition only)

The HTTP ConnectionModule controls the interface between Amber client and server using the HTTP protocol. The properties used by the HttpConnection Module are:

- ConnectionModuleX. Where X is the order in which this module is used starting at 0.

Required. This defines the connection module used. For the HTTP connection module the parameter is "`amber.client.HttpConnection`".

- ConnectionUrl. This property defines the URL used to access the Amber HTTP interface connection servlet. This value can be defined relative to the current HTML server URL. The default value is "/servlet/AmberReceiverConnect".
- SendUrl. This property points to the URL of the Amber HTTP interface which receives packets from the client. The default value is "/servlet/AmberReceiverReceive".
- ReceiveUrl. This property points to the URL of the Amber HTTP interface which sends packets to the client. The default value is "/servlet/AmberReceiverSend".

Appendix: Amber Server Core Functionality

These are the available functions for use with the amber.server.manager.Core object.

- getAmberRoot. This function returns a string containing the path to the Amber server root directory at the server.
- getDocumentRoot. Part of the functionality required by Amber is Web based. Images loaded by image controls, HTML files loaded from within Amber etc are located relative to the Web server document root. This function returns a string pointing to the Web server document root. This allows Amber applications to programmatically generate web content.
- getApplicationManager. This function returns the handle of the ApplicationManager class which is responsible for handling live Amber applications. This allows the programmer to identify/access other running Amber ApplicationHandlers. This could be useful for interconnecting clients such as in a chat program.
- getConnectionManager. This function returns the handle of the ConnectionManager which is responsible for handling incoming connections to the Amber server. It is unlikely that this function is useful to an application programmer.
- getDatabaseManager. One of the most useful functions in the Core, it returns the handle of the Database Manager which controls access to the various databases understood by the Amber server. For more information relating to database access see the section Using Databases below.
- getDeviceManager. This function allows access to the Device Manager system. The device manager system is part of the extended Amber server functionality allowing access and control of remote devices.
- getExtensions. This function returns a Vector of all available extension modules registered with the Amber server.
- getExtension. This function returns the first instance of a specific name in the extension modules.
- getLicenseKey. This function returns the AmberLicenseKey class containing license conditions for this server.
- getLogger. When the Amber server starts it opens a logging file (typically AmberServer.log). This function returns the handle of the Log object responsible for logging errors. Client programs can log information to this object for transmission to the log file.
- getManager. This function returns the ApplicationManager which matches a particular type of incoming connection. Currently Amber understands the following types of connections: Browser, Application, Device.
- getProperties. This returns the PropertyHandler object which controls the AmberServer properties file (config/amberserver/AmberServer.properties). This allows properties to be set in the server properties file and accessed from an application.

- setLoggingLevel. This changes the logging level allowing the amount of information logged by the server to be altered.
- stopServer. Rarely used. This function terminates the server.

Appendix: Basic ApplicationHandler Functions

The ApplicationHandler has a number of support functions available for use. Some of the properties returned by the ApplicationHandler are not valid when the ApplicationHandler is instantiated. See the section Active Parameters for more information. The functions are:

- public Dimension getClientScreenSize(). This function returns the size of the screen at the client computer. This allows the programmer to scale the client components depending on the size of the client screen.
- public Log getLog(). This function returns the log object attached to this application. It is used to log information to the server log file.
- public int getPageId(). Returns the identifier used by the Amber server to work out which ApplicationHandler extended class to instantiate.
- public int getPageSubId(). Returns the secondary identifier used by the Amber server to work out the correct ApplicationHandler to create. This is optional and can be used instead to carry integer information from the Web Server to the application.
- public CoreInterface getParentServer ( ). This function returns a handle to the core server. While this is a CoreInterface object in almost all cases this is likely to be the handle of the central Core class.
- public void logComponentStructure ( ). This function dumps information to the System log on all known objects within the ApplicationHandler. It is used as an aid in debugging.
- public void shutDown ( ). This shuts down the current application. All threads and connections are terminated. This does not shutdown any panels.
- public void shutDownClient ( ). This function tells the client to shutdown. When called a command is sent to the client to terminate.

Appendix: Basic ComponentHandler Functions

These are the basic functions available to all ComponentHandler extended classes. These operations include:

- public ApplicationHandler getParentApplication ( ). This function returns the parent application.
- public ContainerInterface getParentContainer ( ). This function returns any parent container the ComponentHandler is a sub-item of.
- public boolean isApplicationActive ( ). Returns true if the ComponentHandler is connected to the remote client?
- public void forceComponentGetUrl (String urlString). This function forces the remote client browser to get an HTML URL. This would override the current page the Amber client resides in.
- public void forceComponentGetUrl (String urlString, String location). Similar to the other form of forceComponentGetUrl, this function allows the new URL to appear in a different page. Available location strings are:
    - "_self" Show in the window and frame that contain the applet.
    - "_parent" Show in the applet's parent frame. If the applet's frame has no parent frame, acts the same as "_self".
    - "_top" Show in the top-level frame of the applet's window. If the applet's frame is the top-level frame, acts the same as "_self".
    - "_blank" Show in a new, unnamed top-level window.
    - Name Show in the frame or window named name. If a target named name does not already exist, a new top-level window with the specified name is created, and the document is shown there.
- public void setForeground (Color foreground). Sets the foreground color of the control.
- public void setBackground (Color background). Sets the background (fill) colour of the control.
- public void setForegroundBackground (Color foreground, Color background). This allows the programmer to set both foreground and background colours at once.
- public void setFont (Font font). Sets the font of the control.
- public void setEnabled (boolean state). Sets the control to accept/reject input.
- public void setVisible (boolean state). Sets the control as visible/invisible.
- public void requestFocus ( ). Requests that this control get focus.
- public void displayMessageBox (String title, String caption). Displays a window with a caption and a single OK button.
- public int queryMessageBox (String title, String caption, int type). Allows the programmer to ask a question and get a user response. Allowed types are:

    - ComponentHandler.OkType = OK only button.
    - ComponentHandler.OkCancelType = OK/Cancel buttons.
    - ComponentHandler.YesNoType = Yes/No buttons.
    - ComponentHandler.YesNoCancelType = Yes/No/Cancel buttons.

    Returns:
    - ComponentHandler.CanceReturn
    - ComponentHandler.OkYesReturn
    - ComponentHandler.NoReturn
- public void setCursor (Cursor cursor). Sets the cursor to a defined type.
- public void setEventMask (int eventMask). Sets the event mask for the control. This defines which events the control will send.
- public String getComponentStructure ( ). This returns basic information about this control. This is useful in debugging.
- public String getPanelType ( ). Returns the remote client type which corresponds to this ComponentHandler. This information is rarely needed unless the default behaviour of the ComponentHandler is to be altered.
- public FontCharacteristics getFontCharacteristics (...). These functions allow the server to query some of the FontMetric type information on the client. This includes the basic font size information and optionally the width of a string in a specified font.

Also included are all the event handling code which allows the addition/removal of the various listeners supported by Amber. These listeners are:

- Action Listener. (addActionListener/removeActionListener).
- Focus Listener. (addFocusListener/removeFocusListener).
- Item Listener. (addItemListener/removeItemListener).
- Key Listener. (addKeyListener/removeKeyListener).
- Mouse Listener. (addMouseListener/removeMouseListener).
- Mouse Motion Listener. (addMouseMotionListener/removeMouseMotionListener).
- Text Listener. (addTextListener/removeTextListener).
- Window Listener. (addWindowListener/removeWindowListener).

Appendix: Panel Specific Functionality

Panel Drawing Commands

Panels support drawing operations as a series of commands to the panel which are executed in sequence. Later drawing operations may overlap earlier commands. Thus a piece of text could be laid over a filled rectangle. The functions relating to drawing are:

- setBorder. This allows the creation of a border around the entire panel. The border may be raised, lowered or none.
- addXXX. This adds an operation to the sequence of drawing operations. The possible operations are:
    - Draw Image. This allows an image to be drawn at a specified location. The image may be tiled in this case the tiling occurs across the entire panel.
    - Draw String. This renders a string in the current font at the specified location on the panel.
    - Set Colour. The drawing colour used is set to a new value.
    - Draw Line. Draws a line between two points on the panel.
    - Draw Rectangle. A Rectangle is drawn on the screen as specified. The rectangle may be:
        - An empty box.
        - A filled box.
        - An empty 3 dimensional raised or lowered box.
        - A filled 3 dimensional raised or lowered box.
        - An empty box with rounded corners.
        - A filled box with rounded corners.
    - Draw Oval. Draws an oval in the specified location on the panel. It can be filled or empy.
    - Draw Arc. Draws a section of a circle. The arc may or may not be filled.
- removeOperation. This command removes a specified drawing operation.
- removeAllOperations. This command removes all operations specified.

Component Container Functions

The following functions relate to manipulating components on a panel.

- add. This function adds a ComponentHandler to the panel. The syntax of the command is:

  public void add (ComponentHandler newComponent, Object constraints)

  where newComponent is the ComponentHandler to add to the panel and constraints is an instance of amber.type.server.XYConstraints defining the size and location of the control.
  It is the act of adding the component to the panel which creates the visual control at the browser. For this reason if the component is not added to the panel it will not be possible to use it.

- remove. This function removes a control which was already added to the panel. This removes all the visual and messaging elements associated with the control. This is important to note when adding visual Frames (window controls, see below) to the panel. Only when remove is called is the window disposed of. The function syntax is:

  public void remove (ComponentHandler component) throws ComponentHandlerException

  where component is the ComponentHandler to remove from the panel.

- setBounds. This function redefines the X/Y/Width/Height of the control in the panel. Syntax:

  public void setBounds (ComponentHandler component, int x, int y,
  int width, int height) throws ComponentHandlerException

  public void setBounds (ComponentHandler component, Rectangle rect) throws ComponentHandlerException

- setLocation. A subset of setBounds, this function moves the component in the panel. The size of the control is unchanged. Function syntax is:

  public void setLocation (ComponentHandler component, int x, int y) throws ComponentHandlerException

  public void setLocation (ComponentHandler component, Point point) throws ComponentHandlerException

- setSize. The other subset of setBounds this function leaves the component location unchanged but resizes the control. Function syntax is:

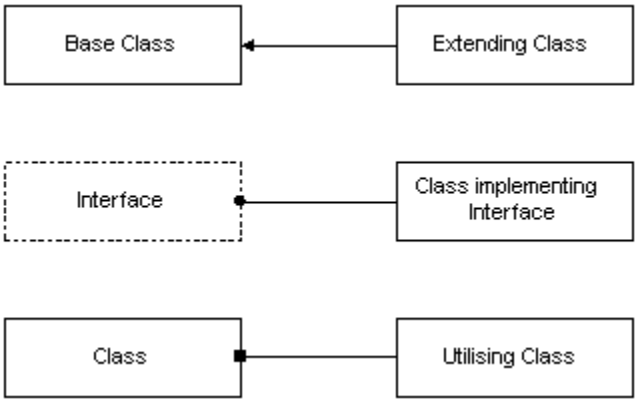
public void setSize (ComponentHandler component, Dimension dimension) throws ComponentHandlerException

public void setSize (ComponentHandler component, int width, int height) throws ComponentHandlerException

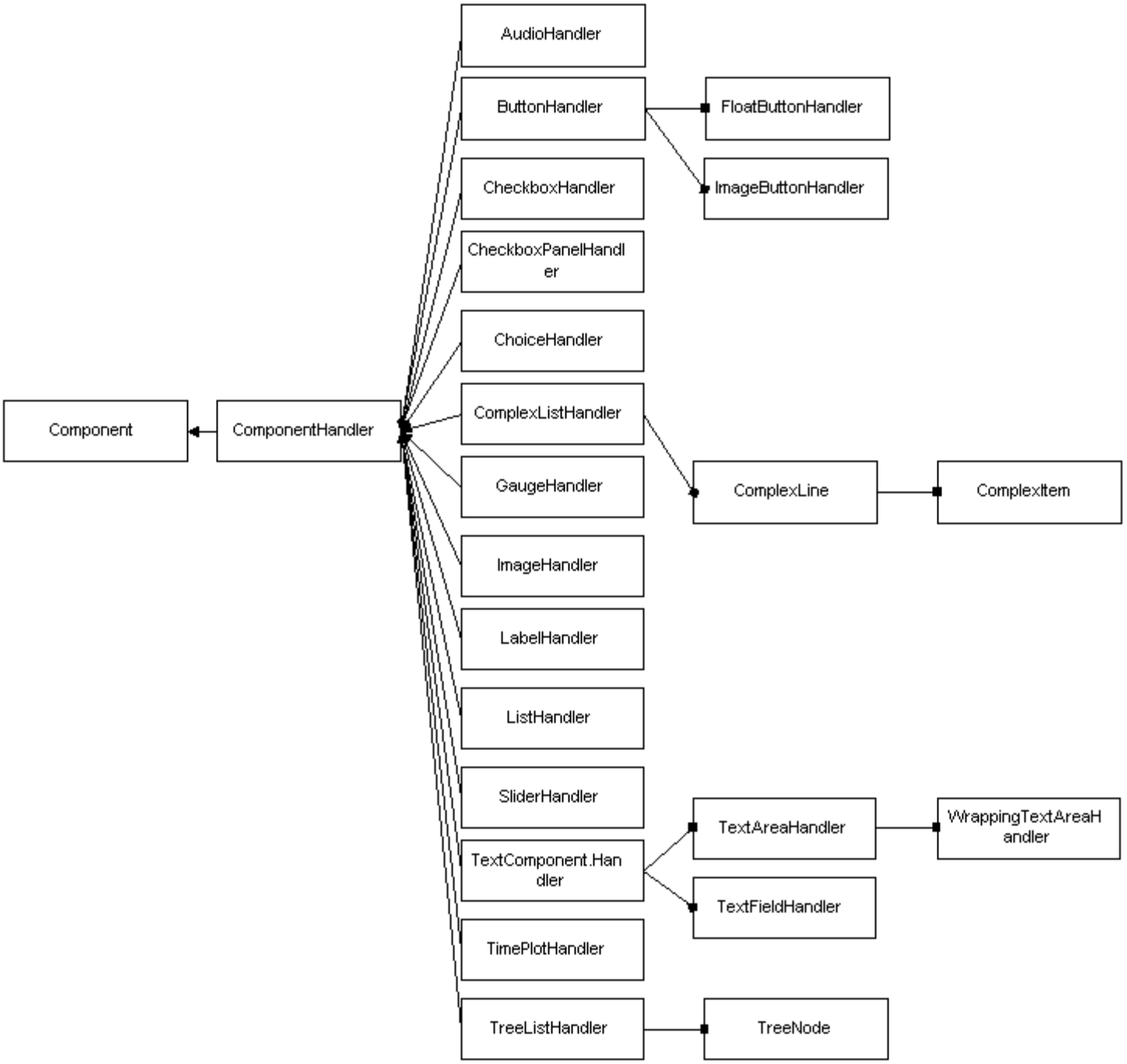
- getInsets. This function returns the amount of the size which is taken up in rendering the control itself. For example, in a Frame the top inset would be the title bar of the window. It is important to note that this information is not valid until the control is created. Thus the information will only be accurate once this has taken place at the client. For example, in the case of a Frame the setVisible function is relatively safe for this information.

Appendix: ComponentHandler Hierarchy

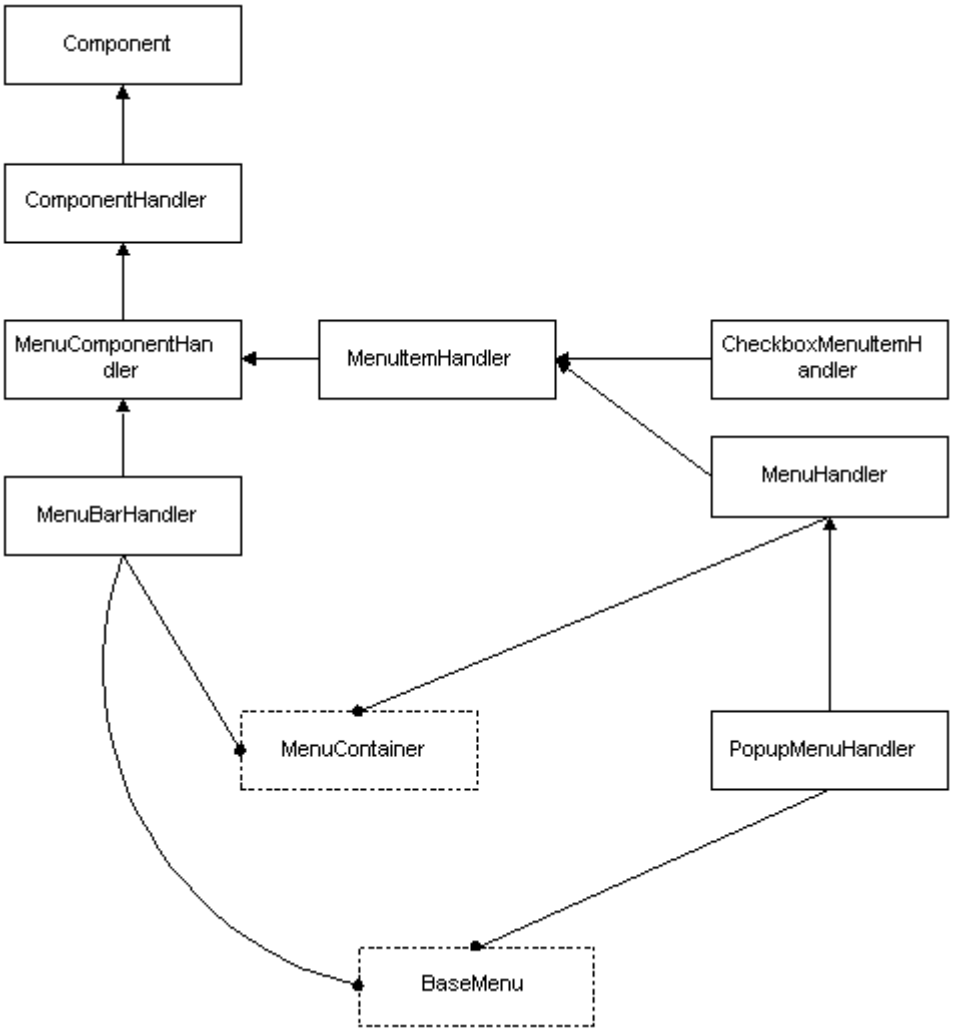
The following diagrams describe the hierarchy of the server component architecture. In these diagrams the following conventions are assumed:



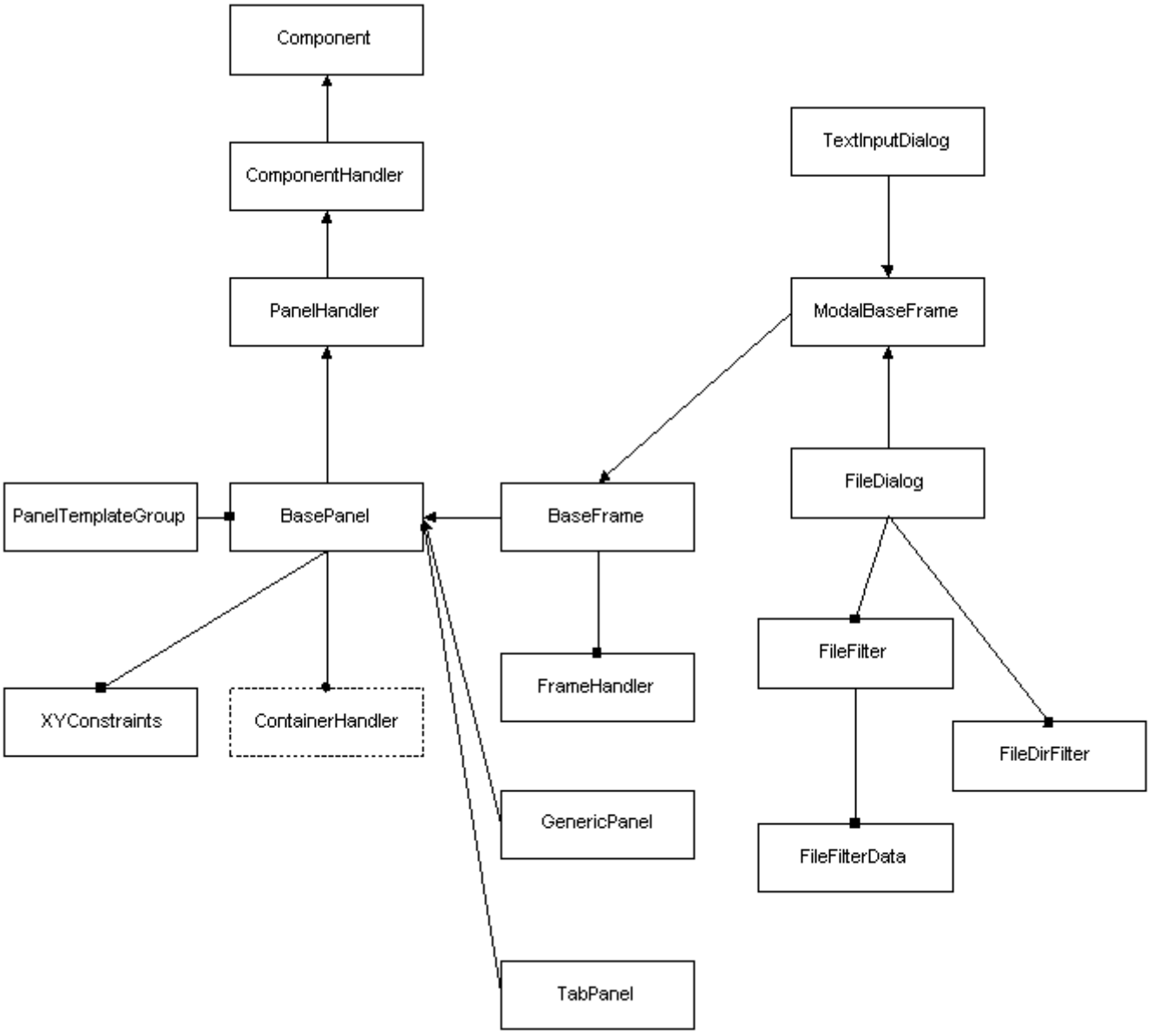
Common ComponentHandler's



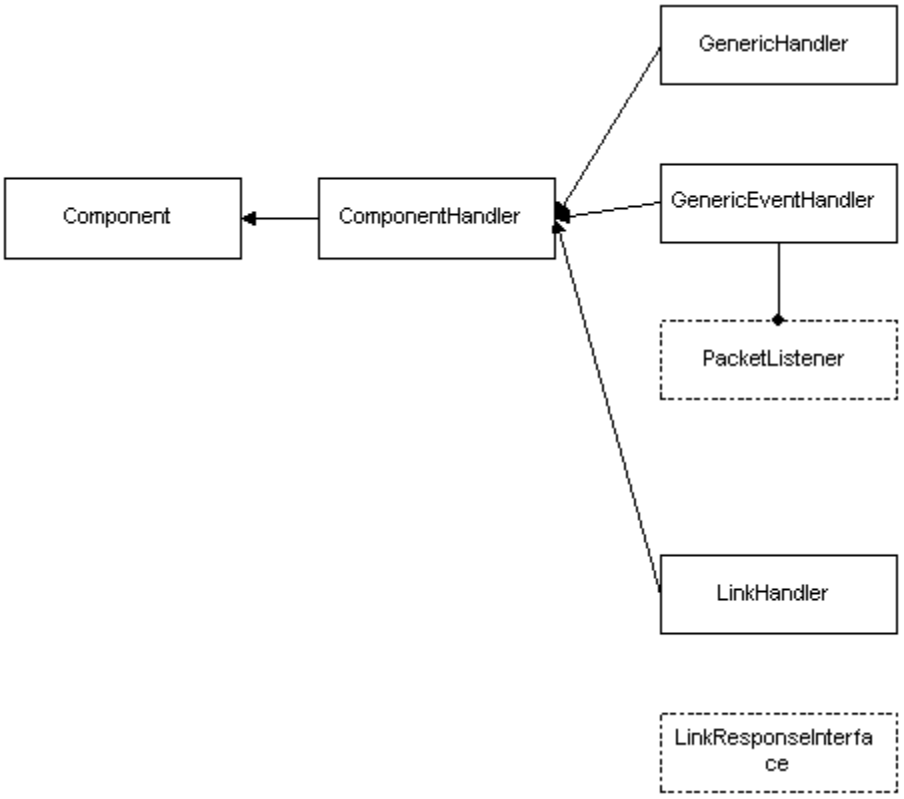
Menu ComponentHandler's



Panel ComponentHandler's



Special ComponentHandler's

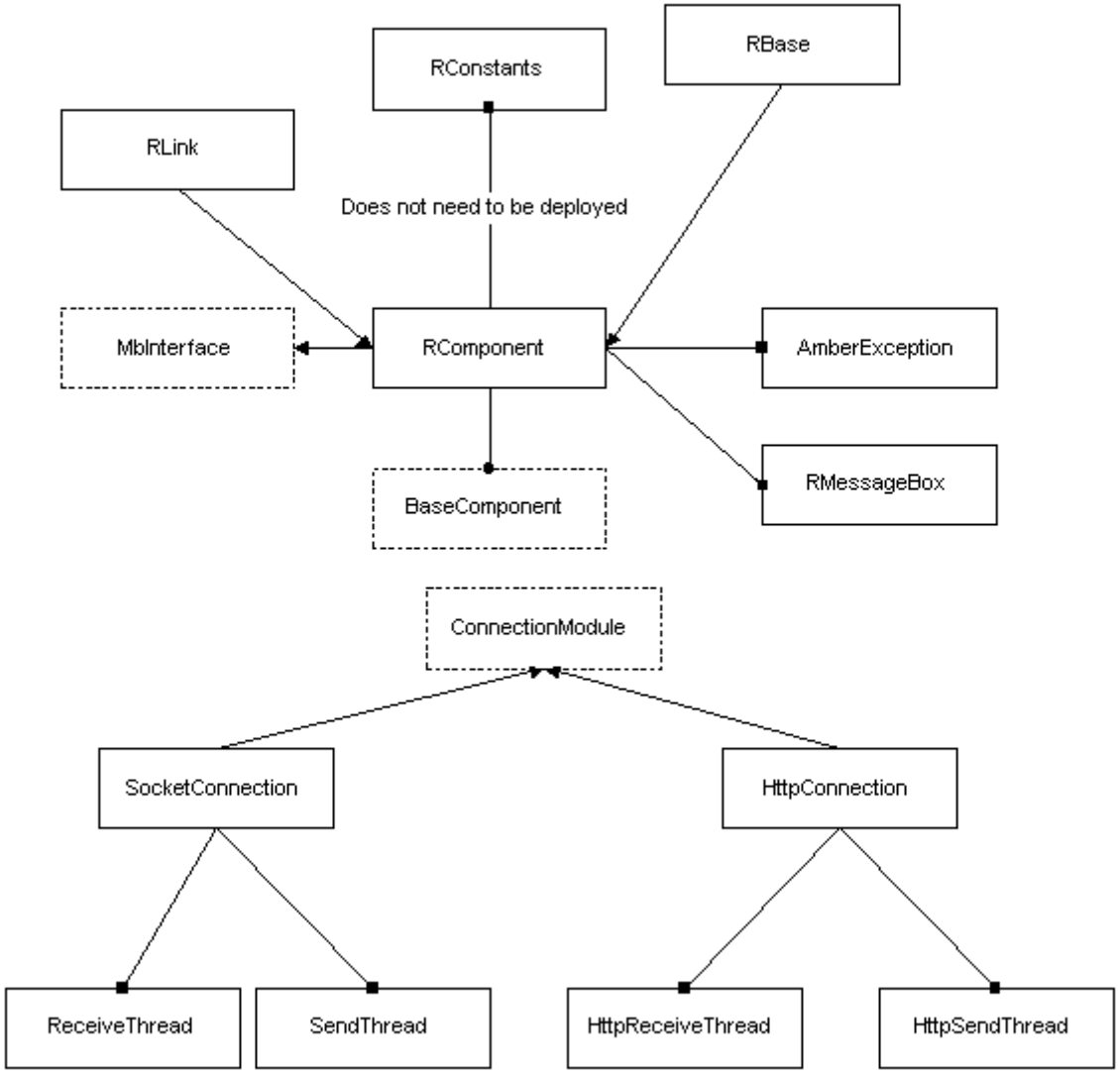


Appendix: Client Component Hierarchy

The following diagrams describe the hierarchy of the components deployed on the client browser. Conventions for the diagrams follow those of previous diagrams.

Basic Client Components

The following diagram shows the hierarchy of the base client components.



Panel Client Components

The following diagram shows the hierarchy of the panel client components.