ULC

Architecture

Guide

RIASUITE
›canoo›

Canoo Engineering AG
Kirschgartenstrasse 5
CH-4051 Basel
Switzerland
Tel: +41 61 228 9444
Fax: +41 61 228 9449
ulc-info@canoo.com
http://ulc.canoo.com

Contents

1 Overview

Canoo RIA Suite is based on ULC technology that provides components to build Rich Internet Applications (RIA) in Java. Use this standard Java library to develop rich, responsive graphical user interfaces (GUIs) for enterprise web applications within Java EE and Java SE infrastructures.

This edition of the *ULC Architecture Guide* accompanies Canoo RIA Suite ULC. It gives a general overview of the architectural setup on the client and server. It is assumed that the reader is familiar with Java programming and has some familiarity with the Java Swing widget set. Since ULC applications are pure Java applications, they can be built using any suitable Java 2 compatible development environment (JRE 1.5 or higher).

Organization

The ULC Architecture Guide is organized into three chapters:

Chapter 2 describes the base framework, the integration of client and server components and the lifecycle.

Chapter 3 provides an in-depth view of the client-side architecture.

Chapter 4 provides an in-depth view of the server-side architecture.

2 Technical Architecture

The following sections introduce the main components of the ULC architecture from a technical perspective. Chapters 3 and 4 provide a more in-depth view of the client- and server-side components, respectively.

2.1 Base Framework

All ULC widgets are split into two halves. The *client half objects* are responsible for the actual presentation of the widget to the user. The *server half objects* are faceless proxies.

The *half object* architecture is implemented by the ULC base framework. The framework includes the *half object classes* implementing the behavior of the half objects and provides the *base infrastructure* (management, communication, etc.).

2.1.1 Base Infrastructure

The base infrastructure manages the lifecycle of the half objects and provides a communication infrastructure for the half object implementations.

Whenever an application creates server half objects (e.g., when a new session is created), the base infrastructure assigns a unique object id to each half object. As soon as the client half objects are required (e.g., for displaying the user interface), the framework automatically uploads the state of the half objects to the client. On the client, this information is used to create the corresponding client half objects and to assign them with the same object ids.

After upload, the ids are used by the infrastructure to dispatch communication issued from a half object to the corresponding half object on the other side. Communication is based on ULC requests, data containers containing the object id of the target half object, a logical name describing the type of the request, and additional request-specific information. On top of this, the half object infrastructure provides a high-level, “remote method call” API for the communication between client- and server-side half objects (see ULC Extension Guide for details on this API).

2.1.2 Half Object Implementations

The server half objects (e.g., *ULCButton*) are faceless proxies for the client-side widgets. A server half object provides an API for the ULC application developer, callback methods to be called by the framework (e.g., for uploading the state to the client), and methods that are called remotely from the client (e.g., for handling state changes on the client). When receiving an event notification (e.g., *ActionEvent*) from the client half, the server half forwards the event to the application event listeners. Whenever an application changes the state of a server half, an appropriate update request is sent to the client.

The client half objects (e.g., *UIButton*) provide the visual representation of the widgets. More precisely, the client half consists of an invisible counterpart to the faceless server half and the visible *basic widget*. The client half object or the basic widget provide methods to handle state change requests provided by the server half. As the client state is updated by user interactions, or the user triggers an event, the client half sends appropriate updates or event requests to the server side.

2.2 Integration

Using a few abstractions, the above discussed ULC base framework is integrated into the client and server runtime environment. Figure 1 shows a top-level view of all main components.

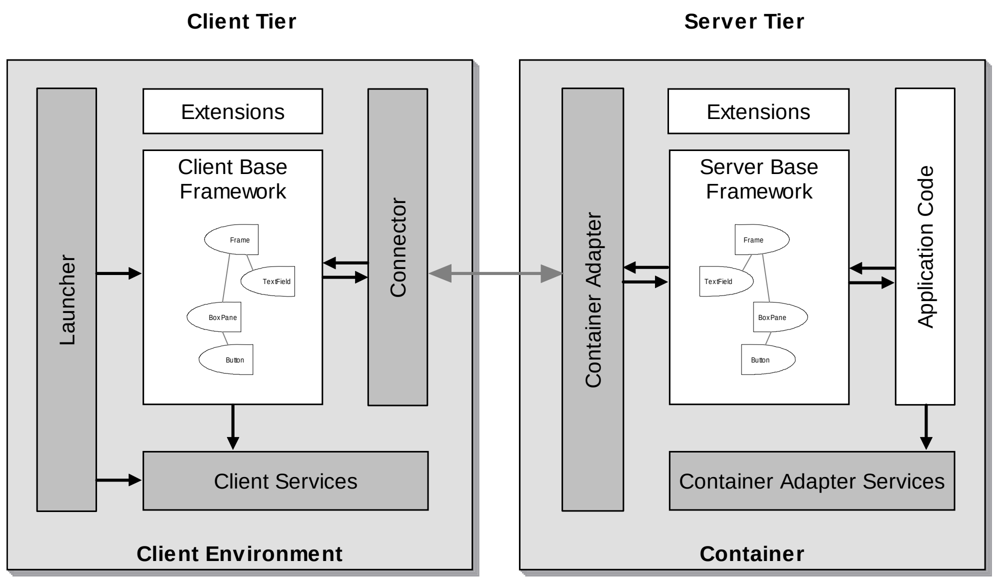

Figure 1: Technical architecture overview

2.2.1 Client Components

The client side consists of the following top-level components: The *client base framework* contains the client-side classes of the base framework. The *connector* implements the actual communication from client to server. Its implementation is directly dependent on the type of container adapter (see below). The connector is used by the ULC client session to send ULC requests to the server session. The *client services* adapt client environment specific services (e.g., file access) to an environment independent interface. These services are used by the other client components. The *launcher* starts and configures the ULC application client components for a specific client environment, e.g., applet in web browser. Optionally, the *extensions* contain application specific code on the client (e.g., application specific client half widgets). A more in-depth description of the client components can be found in Chapter 3.

2.2.2 Server Components

The server side consists of the following components: The *server base framework* contains the server-side classes of the base framework. The *container adapter* integrates the ULC framework into a server runtime container (e.g., servlet engine). The container adapter receives and dispatches communication from the connector and provides server-side services to the framework and to the application. *Container adapter services* provide access to container information and services. The *application* is the actual ULC application implementation that sets up the user interface and registers event listeners to react to user interactions. Optionally, the *extensions* contain application specific extensions to the server base framework (e.g., application specific server half widgets). A more in-depth description of the server components can be found in Chapter 4.

2.3 Lifecycle

A ULC *session* is defined to be a ULC client session (containing the client half objects) and a corresponding ULC server session (containing the server half objects). The lifecycle of all ULC components is tightly coupled with the lifecycle of a ULC session.

2.3.1 Startup

In ULC there are two ways in which a client application can be launched: An application can either be started or resumed.

Starting an Application

To connect to an application from the client machine, the following steps are executed. First, the launcher creates and configures a new client session. The client session then connects to the server side where a corresponding server session is automatically created (by the container or container adapter). After connecting, the client session sends a special initialization request containing information about the client (e.g., screen size).

As a result of the initialization request, the server session calls the *start()* template method of the main application class (implementing the *IApplication* interface), which creates and configures the initial user interface using server half objects. These half objects are then marshalled and sent to the client as a response. Based on this information the client session creates the client half objects.

Resuming an Application

To resume an application a custom application launcher has to send a *resumeApplication* command to the server side. The pause and resume functionality requires both a custom application launcher and a custom container adapter that can support more advanced features for pausing and resuming an application. If an application instance is resumed the complete application instance state prior to pausing the application is reconfigured on the client including the whole editing state (e.g. opened windows and table scroll positions), such that users can continue their work where they left it prior to pausing the application. In contrast to starting an application the connector connects to an existing server-side session.

2.3.2 Runtime

After setting up the initial user interface, the client half objects handle user interactions that may either trigger state changes on the client side or events that application listeners must be notified about. State changes and events are sent to the server session through the connector and container adapter as requests. The server session dispatches these requests to the server half objects that in turn call application code (e.g., event listeners). Application code may in turn change the state of the ULC halves, which results in appropriate responses sent to the client session.

2.3.3 Termination

In ULC there are two ways in which an application can be terminated. In case an application is stopped, it is terminated on the client and the server session is terminated as well. In case the application is paused, only the client is terminated, while the server session is passivated.

Stop Application

In the case of a normal termination of the session (typically as a response to a user interaction) the application terminates the server session. The server session sends a terminate request to the client session.

There are two different scenarios for an abnormal termination of a ULC session:

- In the case of a client error (e.g., exception raised in some client half code) or communication breakdown (e.g., connector cannot connect to container adapter), the client session is stopped immediately. The server session is invalidated and cleaned up.
- If an error condition occurs on the server (e.g., runtime exception raised in application code), the server session sends a *stop* request to the client session, which stops the client session immediately. After sending this request, the server session terminates itself.

Pause Application

To pause an application instance, a custom application launcher has to send a *pauseApplication* command to the server session. On the server side the application will be passivated, and after stopping communication the ULC client will be terminated. The pause and resume functionality requires both a custom application launcher and a custom container adapter that can support more advanced features for pausing and resuming an application.

3 Client Architecture

The client of a ULC application consists of five components:

- The *client base framework*, contains the core framework classes including the client-side implementation of the standard widgets.
- The *connector* is responsible for the communication between the client and server environment.
- The *launcher* integrates the ULC application client into a client environment, e.g., web browser.
- The *client services* adapt the client environment specific services to an environment independent interface, which is used by the other components.
- The *extensions* implement application specific features on the client.

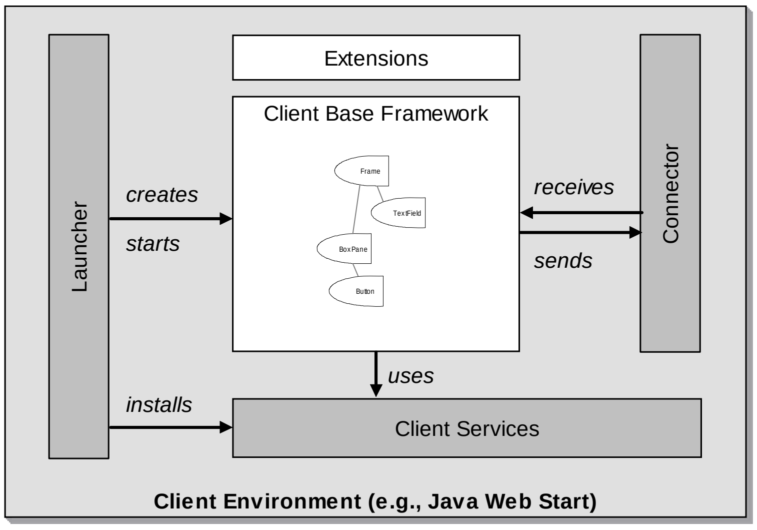

Figure 2: Client technical architecture

Specific configuration of these components may be deployed to clients of ULC applications. This chapter provides the technical background enabling the developer to customize the client deployment to specific needs. A set of typical deployment scenarios is provided in the chapter on *Client Deployment* in the ULC Deployment Guide.

3.1 Client Base Framework

The client base framework consists of components that handle the following aspects:

- **Lifecycle management of a client session:** connecting to server, creating the GUI according to the description received from the server, terminating client session and releasing its resources.
- **Handling GUI events:** handling events locally on the client or forwarding them to the server.
- **Processing requests from the server:** dispatching requests to corresponding client halves.
- **Controlling communication:** sending requests to server and receiving responses.

The client base framework classes can be divided into two categories:

- Core framework classes that provide infrastructure for the client half object implementations.
- Client half object classes that provide the actual implementation of the client half objects.

3.1.1 Core Framework

The following section describes the main components of the client core framework:

- The *UISession* is the main component on the client. It is the root of all other components and is responsible for their lifecycles. The session creates all other components and releases allocated resources on demand. Server requests are dispatched by the session to the corresponding half objects.
- Each session has a *communication controller*, which is responsible for sending requests to and receiving responses from the server in a separate thread. The communication controller ensures that client-server communication is handled asynchronously.
- An *input blocker* component blocks user input while the server handles an event. This behavior is configurable by the application.
- A *service registry* manages client-side service implementations (e.g., browser service) to be used by components of the client base framework or of extensions. The range of services and their implementations may be configured by the launcher and depends on the deployment scenario (see *Client Deployment* in the ULC Deployment Guide).

Starting the Application

The launcher triggers the startup of an application. It therefore creates a connector to the server (e.g., servlet connector), creates a session and passes the connector to this session. When starting the session the following steps are performed:

1. The launcher creates a new session and passes it to a configured connector.
2. Session state listeners are added for handling session events.
3. The communication controller is started.
4. As listener the *DefaultSessionStateListener* will be installed, to handle *sessionStarted*, *sessionStopped* and *sessionError* events.
5. In the *sessionStarted* method the *DefaultSessionStateListener* will start the connector via the session. The connector in turn will issue an initial request to the server side from which the client will receive a session id.
6. Again through the session the *DefaultSessionStateListener* will issue a application start request together with client information.
7. The communication controller takes requests from the request queue and calls a communication method on the session's connector.
8. As a result of this initial request the server sends responses containing commands to build the GUI.
9. The communication controller wraps responses into AWT requests and posts them to the AWT event queue.
10. The AWT main thread takes a wrapped ULC request from the event queue and dispatches it to the session.

11. The session interprets the description contained in the request to build the GUI. For each server half object a corresponding client half object is created and registered in the session’s registry.
12. After creating a half object, its associated widget is created and configured according to its description.

After creating the initial GUI, user interactions trigger server roundtrips, which may cause state changes of GUI components.

Handling User Events and Controlling the GUI

A client half object is associated with each widget that handles events dispatched to this widget. As the code for handling such events is executed on the server side, a server roundtrip is required, sending the requests to the server where the corresponding handler code is executed. If this code changes the state of some widgets (using server half objects), these state changes are returned to the client as responses, where they trigger the actual state changes on the widget. The following list describes the steps that are performed during such a server roundtrip. Assume that the user clicks on a button that causes a label to change its text. The steps are illustrated in Figure 3.

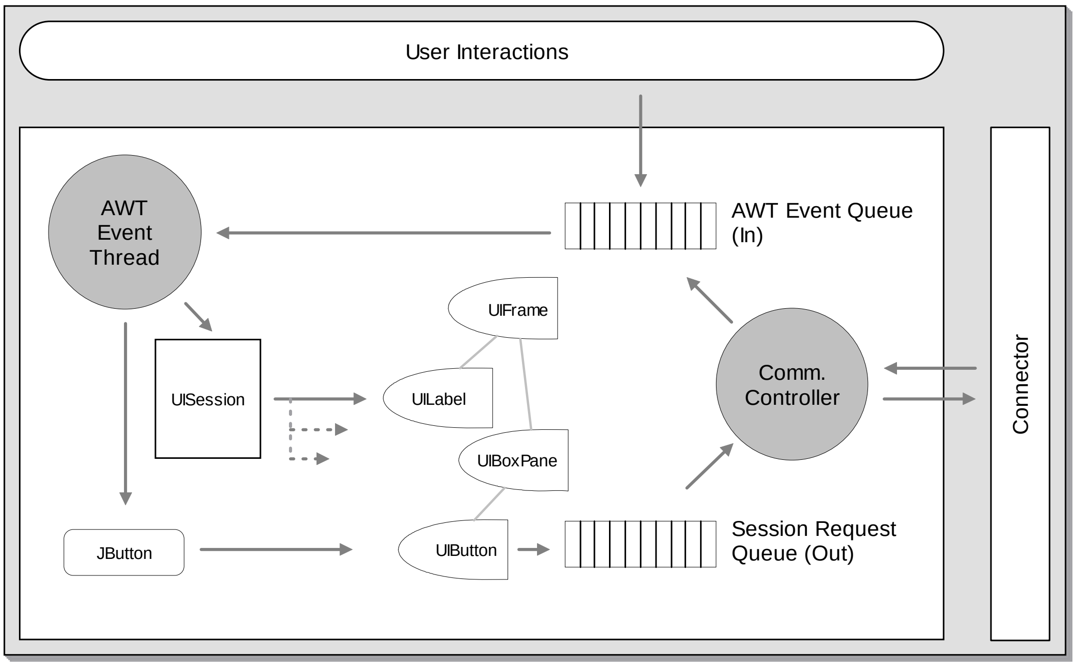

Figure 3: Control flow on client

1. The AWT main thread takes the event (mouse released) from the AWT event queue and dispatches it to the appropriate widget (JButton).
2. The widget notifies the associated half object (UIButton), which is registered as a listener.
3. The half object forwards the event to the server side. This will post a ULC request to the session’s request queue. The request describes the event and includes the id of its originating half object, which is used for dispatching on the server. The handling of the AWT event is finished and the AWT main thread continues executing other AWT events.

4. The communication controller uses the connector to send the requests to the server and to receive potential responses (including the request to change the text of a label).
5. The communication controller posts the received requests to the AWT event queue. A communication cycle is finished and the communication controller proceeds with sending the next requests to the server.
6. The AWT main thread takes a wrapped ULC request from the event queue and dispatches it to the session.
7. The session uses the registry for dispatching the request to the appropriate half object.
8. This will execute the requested commands on the client half object or on the actual widget (e.g., setting its text).

Optimizations

The client base framework implements a set of optimizations to reduce the amount of network traffic. The goal of these optimizations is to enable ULC applications to behave responsively even with low bandwidth and high latency connections. The techniques listed in the following paragraphs are transparent to the application developer. They provide optimal runtime behavior with as little programming attention as possible. Note that in addition to these transparent optimizations, ULC provides a set of APIs, which the developer may use to further improve the responsiveness of the application.

Reducing the Number of Server Roundtrips

- Low-level events like mouse events are translated to high-level semantic events like *actionPerformed()*. Some widgets support such high-level events directly (e.g., *JButton*) for other widgets, the associated client half object performs this translation (e.g., double click on a table cell).
- State updates from client widgets (e.g., window position, contents of a text field, selection state) are not forwarded to the server when the actual state change occurs. A state change of a widget only marks the “changed” property of the widget as dirty. The update of dirty state properties is triggered by the next event sent to the server. This optimization is possible because it is guaranteed that application code (potentially accessing the changed property) is only invoked as part of an event handler. It is therefore sufficient to transfer state updates just before any event handler is called.
- High volume events like *windowMoved()* are not directly forwarded to the server. Nevertheless the corresponding state property will be updated on the server with the next server roundtrip.
- ULC supports the concept of *optional events*, i.e., the client half object knows whether a listener is registered for a specific event on the server side. Events are only forwarded if there is at least one listener registered.
- The communication controller waits for requests to be sent to the server in a separate thread. Often requests come in bulks, e.g., scrolling in a table may trigger several data requests. The communication controller therefore takes all available requests of the session’s request queue and forwards them within a single communication call to the server. The controller waits until the client swing application has handled all current user inputs; which is also the moment in which the AWT event queue will be empty. This wait increases the chance of sending all the requests of a user interaction as one bulk request.

Reducing the Amount of Transferred Data

- As ULC supports lazy loading, not all data of a widget is transferred at initial upload time (e.g., table entries). When data is required which is not yet uploaded, a request is sent to the server. Until the response arrives a dummy value is returned to the requestor. After reception of the requested data, the widget's content is updated. In addition, the data is cached for further use.
- If multiple widgets share the same model, the data is only transmitted to the client once.

Increasing the Responsiveness

- The AWT main thread dispatches events to the components, which may initiate a server roundtrip. As the communication is handled in a separate thread (communication controller) the AWT main thread does not block while waiting for the response. Requests (and AWT events) updating the GUI may be executed in the meantime, e.g., it is possible to resize and hence repaint a window.
- Although the GUI is not blocked while an event is processed on the server, all user input (mouse and keyboard) should be blocked until all responses triggered by the event are processed on the client, potentially disabling GUI components. Blocking all user input during event handling on the server is the default behavior, but can be changed by the application (see *ClientContext.setEventDeliveryMode()*).

3.1.2 Client Half Objects

The major part of the client base framework consists of classes implementing the *client half objects*. They implement the actual visual representation of the faceless server half objects. For each widget contained in the server base framework a corresponding implementation is provided. Client half objects extend the *UIProxy* base class and act as adapters to the actual (Swing) widgets:

- They create and configure the actual (Swing) widget.
- They register themselves as listeners to widget events and forward the appropriate events to the server.
- They can handle requests (remote calls) from the server and perform operations on the actual widget. Most of the time, however, remote calls will be dispatched on the basic widget itself.

The *UIProxy* class provides the following base functionality (for details please refer to the ULC Extension Guide):

- It defines callback methods that are called during the creation of the client half object and its basic widget (*createBasicObject(), preInitializeState(), postInitializeState()*).
- It provides a high-level API for communication with the corresponding server half object. Incoming remote calls from the server are dispatched to the appropriate method on the half object itself or on the basic widget. For communicating back to the server, *UIProxy* offers several methods to issue remote calls (*invokeULC(), updateStateULC(), fireXyzULC()*).

3.2 Connector

The connector is a passive configurable component, which is used by the communication controller to access the server. A connector always corresponds to a container adapter implementation. Currently ULC supports three different connectors:

1. The development connector is used in the development environment to communicate with a ULC application running within the same Java Virtual Machine.
2. The Servlet connector communicates over the HTTP protocol with a servlet container executing a ULC application (see Section 4.2.4).
3. The EJB connector is used to communicate with ULC applications, which are executed as stateful session beans (see Section 4.2.5).

To support pause and resume it is necessary to implement a custom connector (see Chapter 2.3).

A connector must implement the following interface:

```
public interface IConnector {
    public void start() throws ConnectorException;
    public void stop() throws ConnectorException;
    public Request[] sendRequests(Request[] requests)
                                          throws ConnectorException;
}
```

- Before the connector is used the first time the method *start()* is called. If there is a problem accessing the specified server container a *ConnectorException* is thrown.
- Whenever the communication controller sends requests to the server and retrieves the corresponding responses, the connector's *sendRequests()* method is called. The connector must then marshal the requests into a transferable format, send the data to the server, receive the response data from the server and unmarshal this data into responses. Problems which occur at this stage are propagated to the communication controller by throwing a *ConnectorException*.

  Marshalling of objects in a request is handled by an *IStreamCoder* associated with the class of the object being marshalled. The *IStreamCoder* for a class is obtained from a *CoderRegistry* that is provided by a client or server specific *ICoderRegistryProvider*. The connector class is responsible for instantiating the *ICoderRegistryProvider* object from the coder registry provider class name that is specified as an argument to the ULC client launcher. In case the coder registry provider class name has not been specified, ULC's default coder registry provider is used.

  ULC's serialization infrastructure relies on the functionality of the Java classes *java.io.InputStream* and *java.io.OutputStream* to read and write data for communication between client and server. However, in many cases it might be useful to additionally encode/decode the output/input of the ULC serialization infrastructure before the data is eventually written to/read from the raw Java stream.

  IDataStreamProvider is used to obtain *IDataOutputStream* and *IDataInputStream* objects that define operations for writing/reading primitive values and objects in a structured format to some underlying output/input stream. The connector class is responsible for instantiating the *IDataStreamProvider* object from the data stream provider class name that is specified as an argument to the ULC client launcher. In case a data stream provider class name has not been specified, ULC's default data stream provider is used.

In ULC, a carrier stream is the basic Java IO stream which is used to run and encode the higher level communication protocol for exchanging ULC requests and responses. Useful encodings provided by a carrier stream are zipping and base 64 encoding. ULC allows the configuration of the *carrier streams* via *ICarrierStreamProvider* which is used to obtain the InputStream and the OutputStream. The connector class is responsible for instantiating the *ICarrierStreamProvider* object from the carrier stream provider class name that is provided as an argument to the ULC client launcher. In case a carrier stream provider class name has not been specified, ULC's default carrier stream provider is used.

- When a *UISession* is terminated the *stop()* method is called on the connector to allow proper cleanup.

3.2.1 Development Connector

The development connector is used in the development environment, where server and client are executed in the same Java Virtual Machine. Upon startup the development connector creates a server session associated with an application instance. A reference to this session is held that is used for dispatching requests. When calling the method *sendRequests()* on the development connector, the requests are marshalled, unmarshalled, and then dispatched to the server session. The resulting responses are in turn marshalled, unmarshalled and then returned to the caller (communication controller). The development connector is configured with the ULC application class and two optional parameters *initParameters* and *connectionType*. *connectionType* may be used to simulate network delays. The development connector takes the following parameters:

Name	**Type**	**Definition**
applicationClass	String	The application class, implementing the *IApplication* interface.
initParameters	Properties	The init parameters returned by the getInitParameter() method of the ApplicationContext class.
connectionType	ConnectionType	A ConnectionType with a predefined speed.
dataStreamProviderClassName	String	Name of class implementing *IDataStreamProvider*
carrierStreamProviderClassName	String	Name of class implementing *ICarrierStreamProvider*

clientRegistryProviderClassName	String	Name of class implementing *ICoderRegistryProvider*
serverRegistryProviderClassName	String	Name of class implementing *ICoderRegistryProvider*

3.2.2 Servlet Connector

The servlet connector uses the *URLConnection* object returned from the specified URL (*URL.openConnection()*) to communicate to the servlet container hosting the ULC application. ULC requests are posted as HTTP requests to the server potentially receiving ULC responses. The implementation has the following characteristics:

- Since the order of ULC requests must be preserved, only one HTTP request per session is posted at a time
- The *POST* method sends data to the server and receives the responses.
- As ULC requests are marshalled to byte arrays, the content type is set to *application/octet-stream*.
- Within the body of each HTTP request, two integers specifying the session id and the number of batched requests are written to the *URLConnection*'s output stream, and then the marshalled requests are appended.
- The session id is used to distinguish several ULC sessions which may belong to the same HTTP servlet session.
- The ServletConnector is configured with an AbstractRequestPropertyStore. This reflects the different needs for client environments e.g. in an applet opposite to a jnlp environment with respect to cookie handling. The following aspects are handled by a request property store:
  1. *Client-side cookie handling*.
  2. *Proxy authentication*.
- As HTTP sessions may time out, keep-alive requests are sent in configurable intervals in order to prevent the session from being invalidated when a user is inactive for a certain period.
- The process of establishing a *URLConnection* is repeated three times. This mechanism is useful in case of short server overloads where the server may reject connections. This retry mechanism temporarily reduces the risk of a connection breakdown.

The servlet connector is initialized with the following parameters:

Name	**Type**	**Definition**
requestPropertyStore	AbstractRequestPropertyStore	A concrete implementation of the required type
url	String	Specifies the url of the servlet executing the ULC application.
keepAliveInterval	Int	Specifies the interval in seconds between two succeeding keep-alive requests.

		If less or equal to 0 no keep alive requests are sent.
dataStreamProviderClassName	String	Name of class implementing *IDataStreamProvider*.
carrierStreamProviderClassName	String	Name of class implementing *ICarrierStreamProvider*.
coderRegistryProviderClassName	String	Name of class implementing *ICoderRegistryProvider*.
clientConfiguration	IClientConfiguration	Encapsulates the whole ULC application configuration that is set on the client side.
connectorCommandFailureStrategyFactory	IConnectorCommandFailureStrategyFactory	A *IConnectorCommandFailureStrategy* factory.

In addition, a servlet connector can be configured with the following system properties:

Property Name	**Default**	**Definition**
http.proxyHost	none	Specifies the proxy host to be used.
http.proxyPort	none	Specifies the proxy port to be used.

3.2.3 EJB Connector

The EJB connector uses the home interface of the EJB container adapter (see Section 4.2.5) to get the remote interface of the EJB container adapter bean, which is used to transfer requests to the server-side session of a ULC application and to retrieve the corresponding responses. The remote interface of the EJB container adapter bean is defined as follows:

```
public interface EjbContainerAdapter extends EJBObject {
    public byte[] processRequests(byte[] requests) throws RemoteException;
}
```

To get the initial home interface the standard JNDI lookup mechanism is used. The EJB connector can also be configured with an environment property parameter, which is used to create the initial context object (*javax.naming.InitialContext*). Alternatively the required environment parameters can be specified using the appropriate system properties. Using the EJB connector might require provider specific EJB client classes to be deployed on the client. The configuration of the EJB connector highly depends on the EJB client infrastructure used. Please see the ULC Deployment Guide for typical deployment scenarios.

The EjbConnector is initialized with the following parameters:

Name	**Type**	**Definition**
environment	Properties	Properties used by InitialContext

jndiName	String	The JNDI name used for lookup of EJBHome object
dataStreamProviderClassName	String	Name of class implementing *IDataStreamProvider*
carrierStreamProviderClassName	String	Name of class implementing *ICarrierStreamProvider*
coderRegistryProviderClassName	String	Name of class implementing *ICoderRegistryProvider*

3.3 Launcher

The launcher component defines the integration into the client environment. There are four different ways to integrate a ULC application client into a client environment:

- The client may run as an applet in a web browser
- JNLP may be used to start the client.
- A standalone client implementation may be installed, which can be started from the command line or act as a helper application associated with documents with a specific mime type or extension.
- A ULC client may be integrated into an existing Java-based client application using the provided API.

The responsibilities of a launcher are:

- Configuring the client environment adapter.
- Creating and starting connectors.
- Creating and starting client sessions.
- Reacting to error conditions, e.g., inform the user.
- Reacting to session termination, e.g., terminate the client.

3.3.1 Client Environment Adapter Configuration

A launcher is responsible for configuring the client environment adapter, according to the deployment scenario and the needs of the ULC applications to be connected to. The following aspects may be configured:

- If the client is used for ULC applications that need to access the clients file system, an appropriate file service implementation must be installed by the launcher.

  ```
  ClientEnvironmentAdapter.setFileService(IFileService fileService);
  ```

- ULC applications which need to access the client's web browser, either to show a web page through the *ClientContext.showDocument()* API or the help context installed on widgets, require the client to have a browser service installed. The following API may be used:

  ```
  ClientEnvironmentAdapter.setBrowserService(IBrowserService browserService);
  ```

- In the start application request a client sends a set of system properties to the ULC application to be accessed by the *ClientContext.getSystemProperty()* API on the server. By default only the set of standard properties, which are accessible within the default sandbox, are transferred to the server. If other properties are made accessible to the ULC application they must explicitly be set by the launcher using the following API:

```
ClientEnvironmentAdapter.getClientInfo().
                setSystemProperties (Properties props);
```

- Using the *ClientContext.sendMessage()* API, a ULC application may send messages to the client. To react to such messages a client must have the message service installed. This can be done by the API:

```
ClientEnvironmentAdapter.setMessageService(IMessageService messageService);
```

3.3.2 Lifecycle Management

Lifecycle management of client sessions is the launcher's responsibility. In order to react to lifecycle events a launcher usually registers an *ISessionStateListener* on the session. Starting a session consists of:

- Creating a connector
- Creating a client session and passing the connector to it
- Registering a session state listener on the session
- Starting the session

For example the *DefaultJnlpLauncher* creates a *DefaultJnlpLauncher* instance and is started with a configured *ServletConnector*:

```
DefaultJnlpLauncher launcher = new DefaultJnlpLauncher();
URL url = new URL(urlString);
launcher.start(new ServletConnector(new CookieRequestPropertyStore(url),
                url, keepAliveInterval), userParameters);
```

The *start* method is implemented in the *AbstractJnlpLauncher* and handles the default case of starting an application in contrast to pause and resume. A *DefaultSessionStateListener* is installed, then the session is started.

```
public UISession start(IConnector connector, Properties userParameters) {
        UISession session = new UISession(connector, userParameters);
        session.addSessionStateListener(createSessionStateListener());
        session.addSessionStateListener(this);
        session.start();
        return session;
}
```

The following code fragment illustrates the typical startup sequence. To implement the pause and resume functionality for an application a *ISessionStateListeners* needs to be implemented, that will react differently in the *sessionStarted* and *sessionEnded* method.

```
public class DefaultSessionStateListener implements ISessionStateListener {
    public void sessionStarted(UISession session) throws Exception {
        session.startConnector();
        session.sendStartApplication();
    }

    public void sessionEnded(UISession session) throws Exception {
        session.stopConnector();
    }

    public void sessionError(UISession session, Throwable reason) {
        new ErrorDialog("Application Error", "Unrecoverable error.
                    The application will be terminated.", reason).show();
    }
}
```

3.3.3 Launcher Implementations

The standard distribution contains a set of abstract launchers, which implement the basic start sequence and lifecycle management. Abstract launchers are not ready to use and need an implementation to configure the launcher. The distribution also contains configurable default implementations. The following implementations are provided:

Abstract Applet Launcher

The abstract applet launcher is the basic building block for the applet deployment scenario. It is used by the default applet launcher implementation, but may also be used for customized implementations.

Installed Services: The installed browser service uses the API of the applet context to load a web page in a browser. Note that you should always specify the target argument to prevent the web page hosting the UI Engine from being replaced. Otherwise the applet and its associated UI Engine are stopped.

Lifecycle Management: The lifecycle management of the client session is tightly coupled with the applet's (launcher) lifecycle. Calling *start()* on the applet starts the client session, stopping the applet also stops the client session. If the client session is terminated by the application, an empty applet pane is displayed. Error situations are signaled to the user by appropriate dialogs.

Using the AbstractAppletLauncher: To use the abstract applet launcher the template method *createConnector()* must be overridden to return an instance of a connector. To specify the user parameters the applet parameter *user-parameter-names* must contain a comma-separated list of user parameter keys, the appropriate values must be provided with corresponding applet parameters.

Minimal implementation:

```
public class TeamMembersAppletClient extends AbstractAppletLauncher {
    protected IConnector createConnector() {
        int keepAliveInterval = 900; // keep-alive interval in secs
        String url = "http://localhost/TeamMembers",
        return new ServletConnector(new AppletRequestPropertyStore(),
                                    url, keepAliveInterval);
    }
}
```

Configuration:

```
<PARAM NAME = "user-parameter-names" VALUE = "user,role" >
<PARAM NAME = "user" VALUE =" scott">
<PARAM NAME = "role" VALUE ="sales">
```

Default Applet Launcher

The default applet launcher extends the abstract applet launcher with the ability to configure the URL of the ULC application to connect to. It uses the servlet connector to communicate with a servlet container hosting the ULC application. Two applet parameters must be provided to set up the connector, a third for the log-level is optional

Note that it is sufficient to set the log-level to *ALL* to enable client side logging. This turns on ULC's predefined logging facility which only has an impact on the *SimpleLogManager*. Integrating any other logging facility requires an implementation e.g. to handle parameters.

Configuration:

```
<PARAM NAME = "url-string" VALUE = "http://localhost/TeamMembers/" >
<PARAM NAME = "keep-alive-interval" VALUE ="900">
<PARAM NAME = "log-level" VALUE ="ALL">
```

Abstract JNLP Launcher

The abstract JNLP launcher is the basic building block for deployment with JNLP clients. It is used by the default JNLP launcher implementation, but may also be used for customized implementations.

Installed Services: The installed browser service and file service are based on the services defined by the JNLP specification. These services may run within the default sandbox. Whenever this service is used the user gets prompted for granting permissions.

Lifecycle Management: A client session gets started when the launcher's *start()* method is called. Error situations are signaled to the user by appropriate dialogs. When the session terminates the launcher is terminated.

Using the Abstract JNLP Launcher: To use the abstract JNLP launcher a main class must create a JNLP launcher instance and call the *start()* method, passing a connector and user parameters as arguments. To modify the lifecycle management provided by the abstract launcher, subclasses of the launcher may override the appropriate callbacks.

Minimal implementation:

```
public class TeamMembersJnlpClient {
    public static void main(String[] args) {
        AbstractJnlpLauncher launcher = new AbstractJnlpLauncher() {
           // potentially overriding lifecycle callbacks
        };
        IConnector connector =
             new ServletConnector(new CookieRequestPropertyStore(),
                                 "http://localhost/TeamMembers/", 900);
        launcher.start(connector, null);
        }
    }
```

Default JNLP Launcher

The default JNLP launcher uses the abstract JNLP launcher and adds the ability to configure the URL of the ULC application and the user parameters as key value pairs. It uses the servlet connector to communicate with a servlet container hosting the ULC application. The parameter `url-string` is mandatory specifies the URL of the application to connect to. The parameter `keep-alive-interval` is optional and declares the keep-alive interval in seconds. Additional key value pairs may specify other optional parameters for ULC or custom user parameters (use the syntax: *<key>=<value>*).

Note that you can turn on logging on the client side by adding the `log-level` parameter. This only turns on ULC's predefined logging facility (*SimpleLogManager*).

Configuration:

```
    <argument>url-string=http://localhost/TeamMembers/</argument>
    <argument>keep-alive-interval=900</argument>
    <argument>log-level=WARNING</argument>
    <argument>user=scott</argument>
    <argument>role=sales</argument>
```

Abstract Standalone Launcher

The abstract standalone launcher is the basic building block for standalone deployment of ULC application clients. It is used by the default standalone launcher implementation, but may also be used for customized implementations.

Installed Services: The installed browser service uses the *Runtime.exec()* API to start a web browser and display the requested document. This only works on Windows platforms (where the command "*cmd /c start*" with the URL as argument gets executed) but may be adapted for other platforms. The installed file service uses the *java.io* package to implement the required file access. Note that the appropriate permissions must be granted to the launcher in order to enable these services.

Lifecycle Management: A client session gets started when the launchers *start()* method is called. Error situations are signaled to the user by appropriate dialogs. When the session terminates the launcher is terminated.

Using the AbstractStandaloneLauncher: To use the abstract standalone launcher, a main class must create a standalone launcher instance and call the start method on it, passing an instance of a connector and user parameters as arguments. To modify the lifecycle management provided by the abstract launcher, subclasses of the launcher may override the appropriate callbacks.

Minimal implementation:

```
    public class TeamMembersStandaloneClient {
        public static void main(String[] args) {
            AbstractStandaloneLauncher launcher =
                 new AbstractStandaloneLauncher() {
                    // potentially overriding lifecycle callbacks
```

```
                };
            String url =
            IConnector connector =
                new ServletConnector(new CookieRequestPropertyStore(),
                                     "http://localhost/TeamMembers/", 900);
            launcher.start(connector, null);
        }
    }
```

Default Standalone Launcher

The default standalone launcher uses the abstract standalone launcher and adds the ability to configure the URL of the ULC application and the user parameters by command line arguments. It uses the servlet connector to communicate with a servlet container hosting the ULC application. The parameter `url-string` is mandatory specifies the URL of the application to connect to. The parameter `keep-alive-interval` is optional and declares the keep-alive interval in seconds. Additional key value pairs may specify other optional parameters for ULC or custom user parameters (use the syntax: <*key*>=<*value*>).

Note that you can turn on logging on the client side by adding the `log-level` parameter. This only turns on ULC's predefined logging facility (*SimpleLogManager*).

Configuration:

```
java com.ulcjava.environment.standalone.client.DefaultStandaloneLauncher
     url-string=http://localhost/TeamMembers
```

keep-alive-interval=900 log-level=ALL user=scott role=sales

3.4 Client Services

Client services are used to adapt the ULC application client to the environment where it is executed (JNLP, applet or standalone). The set of client service implementations to be installed depends on the chosen deployment scenario and on the requirements of the ULC applications. The following client services may be configured: *Browser Service*, *File Service* and *Message Service*.

3.4.1 Browser Service

The browser service is used to access a web browser on the client's machine (see *Client Access* in the ULC Application Development Guide). This service can be controlled by an application to display documents in a browser, e.g., an HTML page or, if an appropriate plug-in is installed, a PDF document. The second use case for the browser service is the help facility of ULC components.

The standard distribution of ULC provides browser service implementations based on the applet context or the JNLP basic service interface as well as implementations, which use Java's *exec()* API (usable for standalone deployment scenarios).

3.4.2 File Service

The file service is used to access the client's file system (see *Client Access* in the ULC Application Development Guide). A file can be specified either by a relative or full file path, or by popping up a file chooser dialog, where the user can select the appropriate file. The file service may be accessed by an application by using the file upload and file download facilities provided by the *ClientContext*.

File service implementations are provided using the corresponding services of JNLP as well as a variant which is based on *java.io* file access classes (used for example for standalone deployment scenarios). There is no implementation for unsigned applets, if the applet is signed the latter variant may be installed to access files.

3.4.3 Message Service

The message service is used to respond to messages sent from the application using the *sendMessage()* API of the *ClientContext*. There is no default implementation provided in the standard release as there are a wide variety of possible implementations, ranging from controlling client components, starting and controlling other client sessions to interacting with the client environment, e.g., using LiveConnect and JavaScript for multichannel applications.

4 Server Architecture

The server-side classes can be divided into the following four components (see Figure 4):

- The *server base framework* contains the server half object implementations along with the base framework infrastructure such as half object identification and communication.
- The *container adapter* is the adapter to the underlying deployment and runtime container (e.g., servlet engine) making the server base framework and application independent of the chosen container.
- *Container adapter services* provide access to container information and services.
- The actual *application* implementation consists of the application main class (implementation of the *com.ulcjava.base.application.IApplication* interface) and classes it depends on.
- *Extensions* may extend the server base framework to add application specific features (typically widgets).

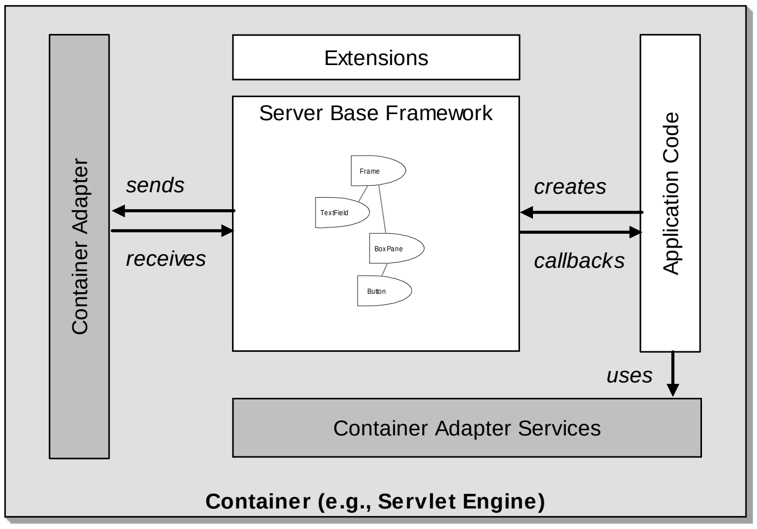

Figure 4: Server technical architecture

This chapter provides technical insight into the server-side implementation and integration of the ULC framework. For normal application development and standard application deployment, knowledge of this architecture is not required. However, this information can prove very useful for a better understanding of the ULC architecture, especially for customizing ULC (e.g., custom container adapter, extensions), or tuning performance in your runtime environment.

4.1 Server Base Framework

The server base framework classes can be divided into two categories:

- Core framework classes that provide the infrastructure for server half object implementations.
- Server half object classes that provide the actual implementation of faceless server half objects.

4.1.1 Core Framework

The central class in the *server core framework* is *ULCSession*. It is the root of all server half objects. This class is responsible for application lifecycle management, management of server half objects, and communication control. Base classes (shared with the client implementation) provide implementations to register half objects, for data containers used for communication, and definitions of constants shared on server and client side. The following sections discuss the responsibilities of the *ULCSession* class.

Application Lifecycle Management

The *ULCSession* class uses the methods of the *IApplication* interface to control the lifecycle of a ULC application. As soon as the *ULCSession* is instantiated, it creates an instance of the application main class. After that it controls the lifecycle of the application as follows:

- The first request received from the client is a special request containing client info. As a result, to this request, the session calls the *IApplication.start()* method on the application main class.
- As the session is stopped the *IApplication.stop()* method is called.
- Before the container passivates a session, the session calls the *IApplication.passivate()* method to give the application a chance to release resources. After the container has reactivated the session, the session calls the *IApplication.activate()* method to allow the application to regain resources.

Note that the lifecycle of the *ULCSession* itself is partially managed by the container adapter, partially by the *ULCSession* itself. See the section on *IApplication* in the ULC Application Development Guide for more detailed information about the different lifecycle events.

Server Half Object Management

As discussed in Chapter 2, all server half objects are assigned a unique identifier (*object id*) they share with their corresponding client half object. Whenever a server class references a half object on the client side (e.g., in a request sent to the client side), this id is used to uniquely identify the respective client half object.

The *ULCSession* class is responsible for providing these unique object ids. Moreover the *ULCSession* class manages the upload process of the server half objects to the client. At session termination, the *ULCSession* class releases all registered half objects.

Communication

The *ULCSession* class is responsible for providing a communication infrastructure for the server half objects. Requests received from the client side are dispatched to the responsible half objects. Responses generated by server half objects are sent back to the client. Note that the dispatching to the correct session is handled by the container (see Section 4.2).

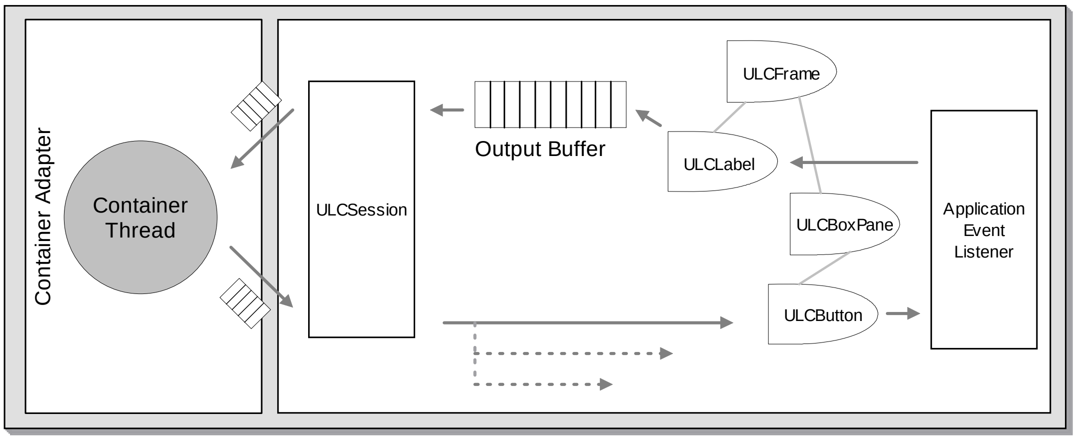

Figure 5: Control flow on server

Figure 5 shows how incoming requests are dispatched to the correct server half objects. Whenever the container adapter (see Chapter 4.2) receives requests from the client, it calls (from within a container thread) the *ULCSession.processRequests()* method, passing the received requests. This method then dispatches the requests one at a time, looking up the target object in the registry (using the object id provided in the request). The server half objects can communicate back to the client. Calling communication API on *ULCProxy* will put corresponding requests to be sent into an output buffer. After processing all received requests, the responses are passed back to the container adapter, which sends them back to the client.

The following example illustrates the above process. It corresponds to the example in Section 3.1.1 where the user clicks on a button, which in turn causes a label to change its text:

The container adapter receives a request containing the description of an action event ("user clicked button"). The request also includes the object id of the originating half object (a *UIButton* instance).

Using information provided by the connector (e.g., a cookie), the container adapter selects the responsible *ULCSession* object and passes the request to this session.

Using the included object id, the *ULCSession* finds the corresponding server half object (a *ULCButton* instance) in the registry and dispatches the request to it.

The *ULCButton* instance notifies registered action listeners about the event.

A listener implemented by the application changes the text of some label (a *ULCLabel* instance) in the user interface. The *ULCLabel* instance calls a high-level communication API method to update the displayed text on the client. This will in turn put a corresponding request into the output buffer of the *ULCSession*.

When all incoming requests have been dispatched, the *ULCSession* takes all requests from its output buffer and passes them back to the container adapter.

The container adapter sends the response back to the client.

4.1.2 Server Half Objects

The *server half objects* component of ULC comprises the implementation of all faceless server half objects acting as proxies for the client half objects providing the actual visual representation. Server half-object classes extend the *ULCProxy* base class and act as a placeholder for the client half objects:

- At upload time, they configure the client half object (or the basic widget itself) appropriately (using the available communication API).
- After upload, they forward state changes to the client side.
- They handle incoming requests from the client side (by providing the appropriate API).

The *ULCProxy* class provides the following base functionality (for details please refer to the ULC Extension Guide):

- It defines a callback method (*uploadStateUI()*) that is called during the upload of the half object to the client.
- It provides a high-level API for communication with the corresponding client half object. Incoming remote calls from the client are dispatched to the appropriate method on a dispatcher object or on the half object itself. For communicating back to the client, *ULCProxy* offers several methods to issue remote calls (*createStateUI()*, *invokeUI()*, *setStateUI()*, ...).

4.2 Container Adapter

The *container adapter* is the adapter to the underlying deployment and runtime container (e.g., servlet engine). This abstraction renders the server base framework and the ULC application independent of the chosen container. Note that some restrictions apply, e.g., with regards to EJB container programming restrictions.

The responsibilities of the container adapter (see Figure 6) are lifecycle management of the *ULCSessions*, receiving requests from the connector and dispatching them to the correct *ULCSession*, and providing a common interface to container services for access from the ULC framework.

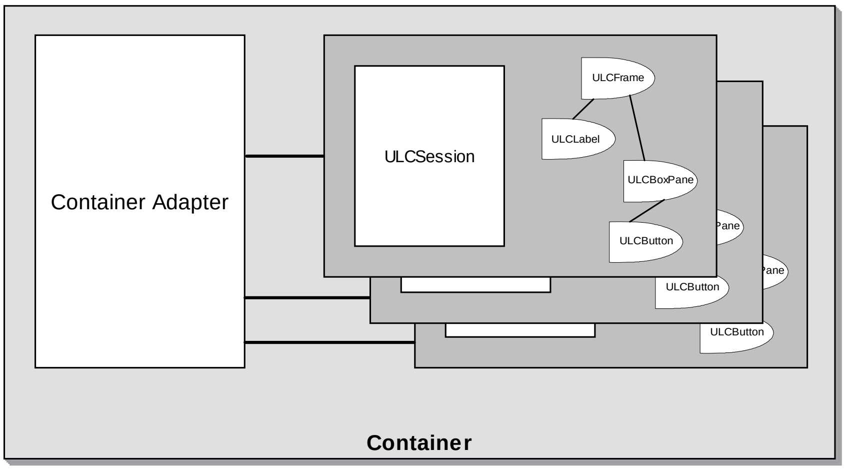

Figure 6: Container adapter

The following sections discuss responsibilities mentioned above and present the two implementations that are part of the ULC release: a servlet container adapter for application deployment in any servlet container (compliant with at least the Servlet 2.4 specification), and an EJB container adapter for application deployment in any EJB container (compliant with at least the EJB 2.1 specification).

4.2.1 Session Lifecycle Management

The container manages the lifecycle and the lookup of *ULCSession* objects:

- When a connector first connects to the container adapter, the container adapter creates a new *ULCSession*. Afterwards, all ULC requests sent by the connector are dispatched to this session which in turn dispatches the requests to the corresponding target half objects.
- The initialization request sent by the client session initiates a call to the *ULCSession.start()* method.
- If the container (adapter) wants to activate or passivate a session, it calls the *ULCSession.activate()* or *ULCSession.passivate()* method, respectively.

4.2.2 Communication

Based on session identification information provided by the connector, the container adapter dispatches incoming requests to the responsible *ULCSession* object by calling the *ULCSession.dispatchRequests()* method. This method then dispatches the requests to the responsible server half objects and returns the responses. The container adapter then sends these responses back to the connector.

4.2.3 Services

The container adapter services provide a common interface for services that container adapters must offer. These services provide access to configuration information (application init parameters) and authentication/authorization information about the connected user.

Moreover, a container adapter can offer services that are specific to the underlying container (e.g., access to the *HttpSession* object in a servlet container). However, the usage of these services in an application limits the deployment to this specific container.

4.2.4 Servlet Container Adapter

The *servlet container adapter* integrates the ULC server-side runtime environment into a servlet engine (compliant to at least the Servlet 2.4 specification). The central component of this implementation is a single servlet. The transport protocol used from the client to the server is HTTP or HTTPS. The session management of the servlet engine is used to dispatch incoming requests to the correct *ULCSession* object.

Figure 7 shows the technical architecture of the servlet container adapter. The various aspects of this implementation are described in the following sections.

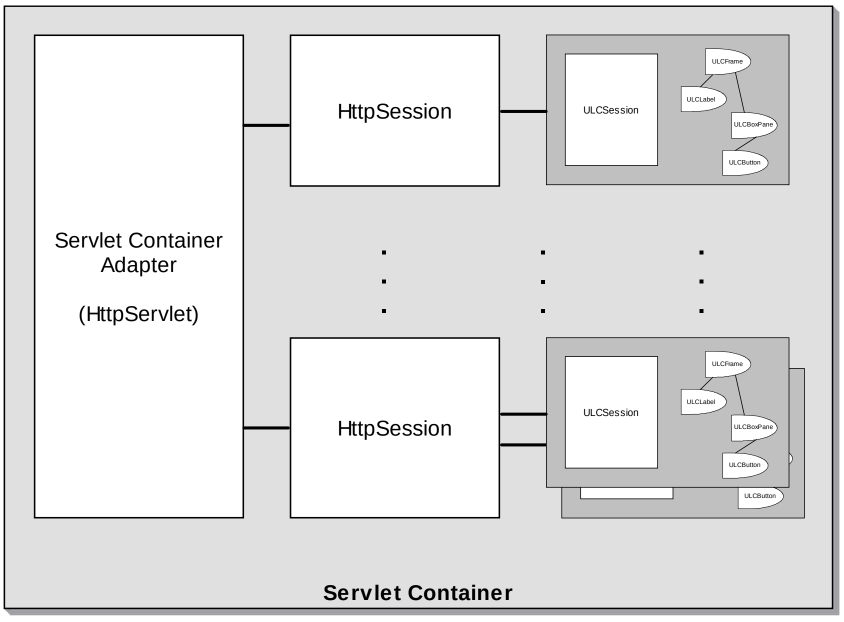

Figure 7: Servlet container adapter

Session Management

For the management (creation and lookup) of server sessions, the servlet container's session management is used. The *ULCSession* object is held in a *javax.servlet.http.HttpSession* object.

As part of the HTTP response, the servlet engine provides a cookie to identify the corresponding *HttpSession* object. The client must then send this cookie along with forthcoming HTTP requests. The client-side servlet connector specifically supports this cookie mechanism.

When running the UI Engine as an applet and using the browser's HTTP(S) infrastructure, the browser handles the cookie management. For this scenario, multiple instances of ULC sessions in one *HttpSession* must be supported (e.g., two browser windows showing two ULC applications as an applet). The servlet container adapter therefore uses an additional ULC specific id for distinction of different *ULCSession* objects in one *HttpSession*. In addition to the *HttpSession* dispatch provided by the servlet container, the servlet container adapter implementation dispatches incoming requests to the correct *ULCSession* according to this id.

Communication

The client-side servlet connector wraps requests in a HTTP POST request. The servlet container adapter wraps responses in a HTTP response. Both request and response contain the following content (with content type *application/octet-stream*):

- internal session id (for distinction of ULC session inside one HttpSession, see above).
- the number of ULC requests contained in the request/response.
- the actual ULC requests (in an optimized ULC specific format).

Container Adapter Services

The servlet container adapter delegates the required container services to the servlet engine as follows:

- The *getInitParameterNames()* and the *getInitParameter()* methods of the *ApplicationContext* class are delegated to the corresponding methods of the *HttpServlet* class. The init parameters for a ULC application must hence be declared as init parameters in the servlet declaration (*<init-param>*).
- The *getUserPrincipal()* and the *isUserInRole()* methods of the *ApplicationContext* class are delegated to the corresponding methods on the current *HttpServletRequest* instance.

In addition to the standard services that the application can access through the *ApplicationContext*, the servlet container adapter provides access to the underlying *HttpSession, HttpServletRequest, HttpServletResponse, ServletConfig* and *ServletContext* objects.

4.2.5 EJB Container Adapter

The *EJB container adapter* integrates the ULC runtime environment into any J2EE-compliant EJB container (at least the EJB 2.1 specification). The implementation uses a stateful session bean for each *ULCSession* object. The session bean holds the *ULCSession* object, receives ULC requests from a corresponding connector, and delegates them to the *ULCSession* object. The transport used by the connector to communicate to the container adapter is specific to the EJB container.

Figure 8 shows the technical architecture of the EJB container adapter. The various aspects of this implementation are described in the following sections.

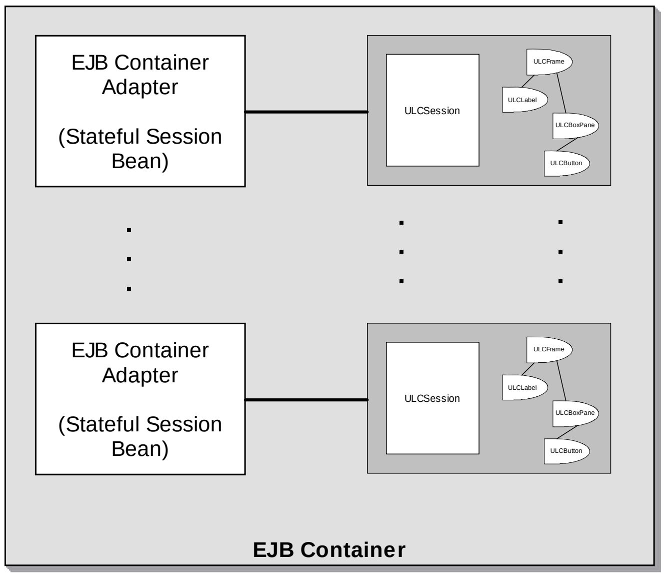

Figure 8: EJB container adapter

Session Management

As there is a one-to-one correspondence of session beans and *ULCSession* objects, the session management is handled transparently by the EJB container infrastructure: Using the session bean's home interface, the connector remotely creates a new instance of the *EjbContainerAdapter* class which in turn creates its *ULCSession*.

Communication

Similar to the session management, the communication aspect is simple to implement for the EJB container adapter: The connector sends ULC requests to the *EjbContainerAdapter* by calling the *processRequests()* method on the remote interface. The bean in turn passes these requests to the *ULCSession*, takes the responses and passes them back to the connector as the result.

Container Adapter Services

The EJB container adapter delegates the required container services to the EJB container as follows:

- The *getInitParameterNames()* and the *getInitParameter()* methods of the *ContainerAdapter* class are implemented through a JNDI lookup in the *"java:comp/env/"* context. Init parameters must hence be declared as environment entries (*<env-entry>*) in the session bean deployment descriptor.
- The *getUserPrincipal()* and *isUserInRole()* methods of the *ContainerAdapter* class are delegated to the methods *getCallerPrincipal()* and *isCallerInRole()*, respectively, of the current *SessionContext* instance.

4.3 Application

The *application* part contains the ULC application implementation, i.e., the application main class implementing the *IApplication* interface and other application classes (views, models, event listeners, business objects, etc.). Note that other parts of the application implementation may reside on other tiers (e.g., EJBs).

The ULC application implementation uses the server half object widgets to create a user interface. Event listeners and data models are provided to these half objects as callbacks to the application. Moreover, the application can use services to access the client (e.g., client file I/O, web browser access) and access information provided by the runtime container (e.g., init parameters).

A new instance of the application main class is created when a new server session is created. The application main class to be used is configured in the respective container adapter.

There are three types of callbacks to an application: lifecycle methods, event listeners, and models.

Lifecycle methods are defined on the *IApplication* interface:

- *IApplication.start()* is called by the *ULCSession* after receiving the initialization request. This is where the application creates and presents the initial user interface.
- *IApplication.stop()* is called after the application terminated (either normally or by some error condition on the client or the server). In this callback the application is supposed to clean up and release all its resources (e.g., DB connections).
- *IApplication.passivate()* is called before the container passivates the server session. In this callback the application is supposed to release any resources (e.g., DB connections) or transient data that should not be serialized.
- *IApplication.activate()* is called after the server session has been activated again. The application should reopen resources or restore transient state.
- *IApplication.handleMessage()* is called when the server session receives a message from the client session.

Event listeners can be registered with the server half widgets by the application to get notified about client-side user events.

Models that are installed in the server half widgets are called back by the framework when data is required on the client. Note that for all these callbacks it is ensured that all half-object states are up-to-date before the callback is executed.

References

[1] Jakarta Tomcat servlet container
http://jakarta.apache.org/tomcat/

[2] Java Web Start
http://java.sun.com/products/javawebstart/

[3] J2EE tutorial on web application archives
http://java.sun.com/j2ee/tutorial/doc/WCC3.html

[4] Eclipse
http://www.eclipse.org/

[5] Jakarta ORO regular expression package
http://jakarta.apache.org/oro/

[6] HTML 4.0.1 specification
http://www.w3.org/TR/html4/

[7] Java Secure Socket Extension (JSSE) 1.0.3
http://java.sun.com/products/jsse/index-103.html

[8] Java Security Architecture
http://java.sun.com/j2se/1.3/docs/guide/security/index.html
http://java.sun.com/j2se/1.4.1/docs/guide/security/index.html

[9] JNLP forum thread describing the protocol handler workaround
http://forum.java.sun.com/thread.jsp?forum=38&thread=71335

[10] Java Plug-In tags
http://java.sun.com/products/plugin/1.3/docs/tags.html
http://java.sun.com/products/plugin/versions.html

[11] Using the conventional APPLET tag with Java Plug-In 1.3.1 and 1.4
http://java.sun.com/products/plugin/1.3.1_01a/faq.html
http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/applet_tag.html

[12] "usePolicy" Permission
http://java.sun.com/products/plugin/1.3/docs/netscape.html#use

[13] LiveConnect
http://developer.netscape.com/docs/technote/javascript/liveconnect/
liveconnect_rh.html

[14] Calling an applet from Java Script
http://java.sun.com/products/plugin/1.3/docs/jsobject.html.

[15] InstallAnywhere
http://www.installanywhere.com/

[16] InstallShield
http://www.installshield.com/

[17] Initializing an initial context with applet parameters
http://java.sun.com/j2se/1.3/docs/api/javax/naming/Context.html#APPLET

[18] Canoo Engineering AG contact address
ulc-info@canoo.com

[19] Introduction to servlet technology
http://java.sun.com/products/servlet/index.html

[20] Servlet specifications
http://java.sun.com/products/servlet/download.html

[21] Introduction to EJB technology
http://java.sun.com/products/ejb/index.html

[22] EJB specifications
http://java.sun.com/products/ejb/docs.html

[23] J2EE Tutorial
http://java.sun.com/j2ee/tutorial/index.html

[24] Packaging J2EE applications
http://java.sun.com/j2ee/tutorial/doc/Overview4.html