UltraLightClient

Rich Thin Clients for J2EE

Technology White Paper

canoo

1 Introduction

The development and the distribution of applications in a rapidly changing business environment is a challenging task. Web applications solve the partitioning and distribution problem. The Web browser as a generic front-end to J2EE applications is now the user interface of choice for enterprise applications on the Internet and Intranet. Unfortunately, it is expensive or even impossible to develop sophisticated rich client user interfaces with HTML. Java's Swing library provides rich client functionality, but its fat client programming model does not easily fit into a J2EE architecture. ULC fills this gap by providing thin client support for Swing, thus enabling a homogeneous, server-side J2EE architecture that allows mix & match of HTML and rich clients.

2 Management Summary

UltraLightClient (ULC) is a component library for developing rich client J2EE applications. It complements HTML based Web applications by enabling cost-effective development of applications with sophisticated and highly interactive graphical user interfaces (GUIs) for the Intranet and Internet.

The J2EE architecture blueprint propagates server-side computing. The intention is twofold: 1) a server-side architecture simplifies development because the difficult issue of software partitioning between client and server is resolved in a generic way, and 2) less software on the client side results in lower software configuration and distribution cost. Fat clients are one option for J2EE clients. Unfortunately, fat clients leave the partitioning of the software architecture to the developer. They must take care of issues such as server-roundtrip optimizations, caching, client-server synchronization etc. In addition, a client-side software update is required for each release.

Web browsers are the most popular J2EE clients. Servlets and additional frameworks such as Struts[1] provide a server-side programming model, which relieves developers from some of the burdens of partitioning like client-server synchronization. This works perfectly for simple, form-based user interfaces but does not scale for sophisticated user interfaces. Throwing in additional technologies like JavaScript and XML/XSLT may result in somewhat improved user interfaces. Yet, these technologies complicate the developer's task and increase development cost. Even worse: substantial use of JavaScript leads to a fat client with undesirable browser dependencies.

ULC resolves the issues mentioned. It extends the capabilities of Swing in such a way that the ULC enabled Swing components fit into the server-side programming model of Web applications. ULC is entirely based on standards such as Java, Swing, J2EE, HTTP(S), or RMI/IIOP. This is great news for Web application developers. Now they can develop rich clients based on a homogeneous and comprehensive API as known from Swing.

ULC affects only the presentation layer within a J2EE architecture and therefore complements HTML perfectly in a multi-channel scenario. The ULC component library relieves the developer of technical issues such as partitioning, optimizing client-server communication, and caching.

A ULC based application can be deployed as a Servlet in a Web container. Alternatively, it may also be deployed as a stateful session bean, profiting from the more advanced management features of EJB containers.

The client side of a ULC based application is handled by a generic presentation engine. This piece of Java code can be distributed as an applet, via Java Web Start, or as a standalone application. Communication usually runs over HTTP(S) but is configurable and may be replaced by other protocols to meet different requirements for security, efficiency, or container handling.

3 Client-Server Computing – Partitioning and Distribution Challenges

Personal computing started in isolation. The development of local and wide-area networks paved the way for client-server computing. The client part of this distributed architecture has usually been a fat client. This allows for robust, sophisticated user interfaces and usage of the substantial local resources (CPU, memory, etc.) of today's desktop computers. Unfortunately, fat clients have several serious drawbacks:

- The client and server part of an application have to be kept in sync, both during development and in production. Since the application logic is partitioned between at least two tiers of the software

[1] http://jakarta.apache.org/struts/

architecture, the business model has to be implemented on different platforms, possibly using different paradigms, programming languages, and operating systems. The lack of standardized infrastructure for such partitioning typically leads to project specific reimplementation.

- All middleware components needed by an application have to be bundled with the fat client. Apart from increasing the distribution load, this significantly complicates the configuration of clients. It also leads to an ever-expanding integration and testing effort. At the same time, client platforms become more and more fragile due to a multitude of components.
- Each application has to be distributed to all clients. A large number of clients results in a costly software distribution infrastructure. With rapidly changing business requirements, this infrastructure will be an impediment for a business-driven release cycle.

The Internet has introduced two additional challenges. Firstly, local application installation is severely hampered due to heterogeneous unmanaged platforms, lack of user knowledge, and security restrictions. Secondly, limited and unpredictable network bandwidth necessitates optimization of client-server communication. It also prohibits frequent distribution of multi-megabyte software packages.

3.1 Partitioning

From a software architecture perspective, client-server applications are a distributed system. As with any such system, a software architect has to decide about the partitioning, i.e. where to cut the system into subsystems (e.g. components, layers, etc.), and where to locate these subsystems physically. A client-server application is typically split into three major software layers, viz. for presentation logic, business logic, and business objects. The presentation layer takes care of building the user interface and handling user events. The business logic layer deals with business rules. Finally, the business object layer contains the basic business entities and takes care of persistence. Partitioning an application between presentation layer and business object layer may seem to be a good choice. However, it is not: this partitioning requires a client-side development model for the presentation layer, and a server-side development model for the other layers, which leads to complexity and numerous pitfalls.

Ideally, partitioning is taken care of by the architecture and a corresponding software library, such that the developer does not have to worry about it. This can only be achieved by a generic partitioning approach. Such a generic partitioning is feasible, but only within the presentation layer, because this is the only layer whose objects – viz. user interface components - are of limited variety and defined in advance.

Since partitioning is probably the most important issue in software architecture, we will have a closer look at the most popular approaches followed to tackle the problem.

3.2 Approaches

There are three major approaches to partition a client-server application, namely *fat client*, *display server*, and *presentation server*.

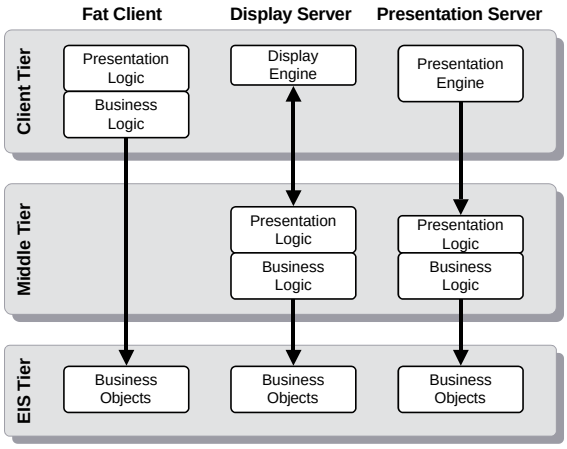

Fig. 1 Fat client, display server, and presentation server

The *distribution server* approach (e.g. Java applets[2] or Java Web Start[3]) does not support any partitioning per se, but takes care of the physical distribution of software. It therefore complements the approaches mentioned above.

[2] http://java.sun.com/applets/index.html
[3] http://java.sun.com/products/javawebstart

3.2.1 Fat Client

A fat client combines presentation logic and at least part of the business logic on the client. Therefore, the architecture is partitioned either within the business logic layer or between the business logic layer and the business object layer.

The fat client can provide a rich user interface experience including direct manipulation, immediate feedback etc. In addition, a fat client may use the local resources of today's PCs, such as processing power. As a consequence, a fat client application behaves similarly to off-the-shelf productivity applications.

From a development perspective, a fat client is rather challenging. Partitioning within the business logic layer requires a developer to keep the client and server parts in sync, both at development and run-time. Alternatively, if the entire business logic layer is implemented on the client, this leads to high network traffic and little reuse of business logic. In any case, sophisticated caching must be implemented to achieve acceptable performance.

Additionally, with every change, the fat client has to be redeployed on every user's PC. The physical distribution can be handled with a distribution server. However, the configuration of a client workstation often turns out to be really complex because each fat client typically requires different versions of infrastructure libraries for communication, database access, etc.

3.2.2 Display Server

Display servers run the entire application on the server. The client merely displays the graphical output of the application and forwards user input like key strokes and mouse movement back to the application. The application's interface to the client is on the level of drawing and input primitives. Popular products are X Window System[4], Citrix MetaFrame[5], and Windows Remote Desktop[6]. Character-based terminals are also an example of the display server approach, but they are not discussed further since their user interface is not state-of-the-art. The X Window System requires usage of specific libraries for development, whereas MetaFrame and Remote Desktop are transparent from a development perspective. Almost any Windows application can be installed and operated under MetaFrame or Remote Desktop.

Client-side software configuration is simple with display servers. Installation of the Display Engine is all that is needed. A further advantage is that MetaFrame or Remote Desktop have no impact on development or software architecture. Applications may be developed as fat clients, maintaining a high user interface standard. Legacy applications or off-the-shelf applications can be integrated easily. An additional bonus is that display server clients are available for a variety of platforms. The disadvantages of the display server approach are located in the area of enterprise computing requirements: scalability is typically limited, due to the fact that conventional desktop applications are not designed to scale in a server environment. Further show stoppers may be the network bandwidth required, latency and firewalls that exclude communication with the protocols employed. Furthermore, server configuration can be difficult if different applications call for potentially incompatible middleware to connect to backend systems.

3.2.3 Presentation Server

Presentation servers run business and presentation logic on the server, but leave management and display of user interface widgets as well as local input handling to the client. The presentation part manages the user interface locally as far as possible, in order to minimize server round trips. The client is (conceptually) stateless and the user session state is maintained entirely on the server.

In the same way as for display servers, the client side consists of an application independent presentation client only. Since applications are specifically developed for server side operation, configuration is simple and scalability is excellent. The disadvantage of this approach is that legacy or off-the-shelf applications cannot be deployed without modification. Typically, the user interface part of such applications has to be reengineered, using specific presentation server libraries such as the Java servlet framework or Struts.

[4] http://www.x.org
[5] http://www.citrix.com
[6] http://support.microsoft.com/default.aspx?scid=kb;EN-US;q186607

The most well-known examples of the presentation server approach are Web browsers connected to application servers that dynamically generate HTML. Despite its popularity, this solution has several shortcomings:

- With HTML, the user interface is mediocre and resembles an old-fashioned terminal application. The former vision of a business desktop, serving as an integrating hub for several applications, has been replaced by the inferior portal metaphor.
- The development of HTML-based applications is expensive due to a heterogeneous environment involving not only HTML and Java, but further technologies like JavaScript, CSS, DHTML, XML, and more.
- Extensive usage of JavaScript to overcome the limitations of HTML has negative architectural implications because it moves application dependent code to the client. This leads to fat client development. Moreover, it restricts browser compatibility.
- HTML consumes considerable network bandwidth if used for complex user interfaces, limiting scalability. For each (possibly tiny) update of the user interface, the entire HTML page has to be sent across the network.
- Interactive applications require a bidirectional protocol. HTTP(S) is only unidirectional (i.e. request-response) and therefore not suited for interactive user interfaces.

Despite their serious drawbacks, HTML-based applications have become increasingly successful. This success is clearly a consequence of the substantial advantages of the presentation server approach, which is, incidentally, the choice favoured by the J2EE architecture blueprint.

3.3 Conclusions

As shown above, architectures based on the presentation server approach are the best choice for new applications both in the Intranet and Internet. HTML-based platforms using servlets or Java Server Pages have been the most widely used solution following this approach, despite their serious disadvantages.

These insights have led to the development of ULC. ULC leverages the advantages of the presentation server approach while avoiding the disadvantages of HTML and JavaScript.

4 UltraLightClient

UltraLightClient (ULC) is a library that adds rich client functionality to thin client J2EE applications, following the presentation server approach. Specifically, ULC provides thin client support for Swing, the standard component library for rich Java clients. ULC supports a homogeneous, server-side programming model, i.e. the developer can use the ULC-enabled Swing components as if they were running on the server. Deployment is possible in any J2EE-compliant[7] container as a servlet or a stateful session bean. On the client tier, ULC employs an application independent presentation engine based on J2SE[8]. Both the distribution of the presentation layer and network optimizations at runtime are transparent to the software developer.

4.1 ULC Architecture

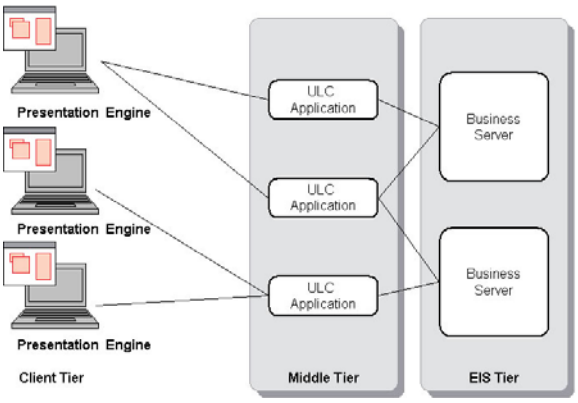

Fig. 2 ULC architecture

ULC follows the presentation server approach described above and therefore has a generic Presentation Engine. The latter is implemented in Java and uses the Swing library for displaying the user interface and handling user input. The Presentation Engine is fairly small (~500 KB) and can be run as a Java application or a Java applet. It is stateless, i.e. the entire presentation state is maintained on the server side, though the state is cached on the client in order to improve performance. A single Presentation Engine will act as the client installation for any number of applications running on any number of servers.

The ULC server side consists of a library that comprises a widget set and support for J2EE container integration. Using the latter, a ULC-based application can be deployed in any J2EE-compliant container, either as a servlet or a stateful session bean.

ULC comes with the same widgets as Swing and the application programmer interface strongly resembles the API of Swing, though the widgets have been streamlined for remote operation and minimal network traffic. Experience has shown that developers who are familiar with Swing become productive in almost no time at all. For a developer, the distributed architecture is transparent, i.e. the issue of partitioning between client and server is handled by the library.

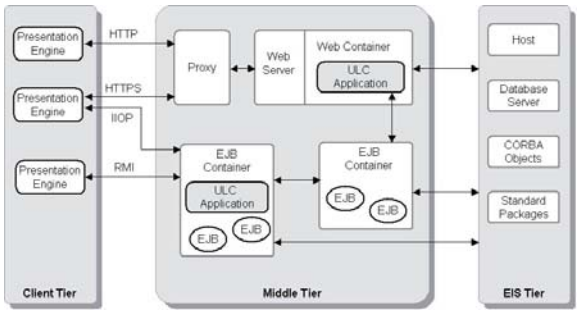

Fig. 3 Architecture of client server applications with ULC

Like all Presentation Server approaches, ULC partitions the presentation layer of an application. Each ULC widget consists of a client-side and a server-side implementation. These implementations are called “half objects”. The client-side half object is responsible for display, user interaction, and event forwarding to the server. The server-side half object represents a proxy to the client-side half object. It does not have a graphical representation, is optimized for operation within an application server, and is therefore called a “faceless” widget.

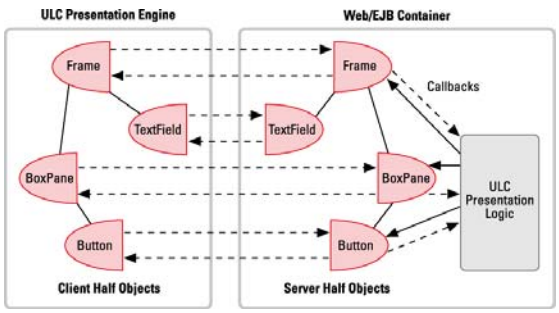

Fig. 4 ULC “half objects”

[7] http://java.sun.com/j2ee
[8] http://java.sun.com/j2se

User interface development is based on the API of the faceless widgets. The communication and synchronization between client-side and server-side half-objects is handled transparently by ULC.

At application start-up, a description of the visible parts of the user interface - as implemented with the faceless widgets - is transmitted to the client. Based on this description, the Presentation Engine builds the user interface, using the corresponding client-side half objects.

4.1.1 View and Model on the Server

Typical ULC based applications follow the well-known Model-View-Controller approach. The view is split between client and server by using the ULC half objects. The integration between presentation layer (view) and business logic (model) stays on the server. As a consequence, the business model is not partitioned between client and server, but resides entirely on the server. This eases development and maintenance of the business object model substantially. In addition, this helps to economize on network bandwidth.

Notice that ULC affects only the presentation layer of an application. All options for the other layers are open, i.e. a ULC based application can take advantage of the full range of services available in a server-based J2EE platform. Server-side presentation objects, business logic and business objects may be handled within a single container or partitioned as in a service-oriented architecture.

4.2 Comprehensive Widget Set

ULC provides a comprehensive set of basic widgets: menus, push buttons, radio buttons, check boxes, combo boxes, text entry fields, text areas, lists, labels, and sliders. In addition, enhanced widgets like tabbed panes, tables, trees, table trees, split panes, and tool bars are available, as well as drag and drop functionality. The widget API is based on the API of Swing because the Presentation Engine uses the corresponding Swing widgets for display and interaction.

Data-centric widgets, like tables or trees, can be challenging if they require extensive data copying. For example, data is first read from a database into a table model and then copied into internal structures of the table widget again. ULC avoids such inefficiencies by providing an adapter model, which allows the developer to define how the table widget can access data directly.

ULC is designed to be visually appealing in different target environments. It offers the full versatility of Swing and an additional descriptive layout mechanism that is simpler to use than Swing's GridBagLayout or BoxLayout.

4.3 Optimized for Low Bandwidth

4.3.1 Lazy Loading and Caching

ULC minimizes network traffic between client and server. This is done transparently, but, if necessary, can be configured by the developer.

The Presentation Engine requests data from the server only if it needs this data to display or update the user interface. Moreover, data is cached by the Presentation Engine, which further reduces network load. A typical example is a table view that shows a small subset of a large table model. Non-visible rows are transmitted only if the user scrolls the table. Such lazy loading is available for tables, trees, tabbed panes, lists, and windows.

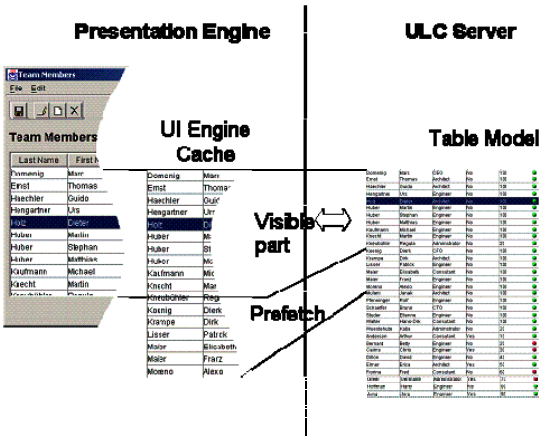

Fig. 5 Table view and lazy loading

4.3.2 Local User Interface Management

Formatters, validators, and enablers are other means of reducing server round-trips. Formatters and validators are used when the contents of an input field can be checked locally on the client. ULC provides ready-to-use validators such as date validators or range validators. More complex syntactic checks are feasible by using the regular expression validator. Enablers can be attached to widgets and can change the state of this widget based on the state of some other widget. For example, a button can be disabled or enabled depending on an input field being empty or not. This is also executed locally without server round-trip.

4.3.3 Event Handling

ULC further optimizes server round-trips by sending only those events to the server that have an appropriate event handler registered with the server-side half object.

4.3.4 Asynchronous Communication

Applications that rely on a server connection must cope with the fact that a server round-trip may take any amount of time. Reasons for delays may be server overload, high network latency, temporary network failure etc. Particularly on the Internet the response time can vary greatly. For a user it is annoying when the user interface is blocked in situations like this. Though blocking cannot be avoided everywhere, ULC's capability for asynchronous communication helps to alleviate the issue. Lazy loading, for example, is performed asynchronously: while waiting for the data, ULC does not block the user interface.

In order to improve the user interface experience, events can be delivered asynchronously as well.

Furthermore, multiple requests can be bundled in blocks for transmission. This optimizes client-server communication by avoiding the overhead of numerous small-sized requests.

4.4 ULC Development

ULC preserves the AWT/Swing programming model as far as possible. For example, event handling is performed by delegation to listeners that have previously been registered with the widget; layout resembles that of the GridBagLayout, etc. As a result, a ULC application is developed in a fashion similar to a traditional Java fat client. The essential difference to fat client development is that the programming model is server-side, i.e. the developer will use the ULC widgets as if they were running on the server. For user interface design, a point and click visual editor is available, based on the Eclipse IDE[9].

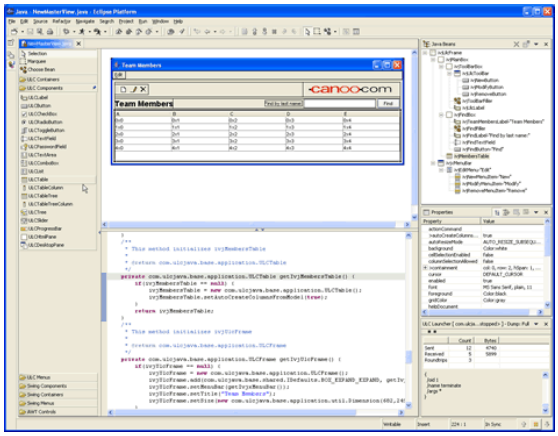

Fig. 6 Screenshot of ULC Visual Editor for Eclipse

To expedite edit-compile-execute-cycles, ULC provides a client / server simulator that offers the option to run the Presentation Engine and the ULC based application within the same Java VM. In this way, deployment of code in a J2EE container can be avoided. Moreover, this simulator allows the developer to set up testing scenarios with varying network bandwidth and latency.

The server-side programming model means that the user interface of a ULC based application is developed as if it were running on the server. Since ULC based applications are typically multi user applications with hun-

[9] http://www.eclipse.org

dreds or thousands of concurrent clients, some restrictions regarding concurrency, synchronization or activation/passivation must be observed. For this purpose, ULC provides facilities to make use of the corresponding container services.

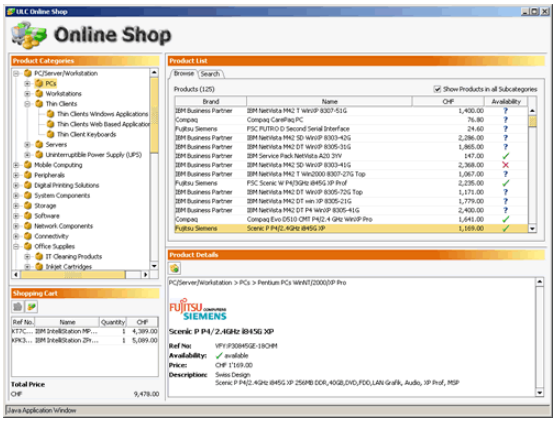

Fig. 7 Screenshot of example application "Product Catalog"

The ULC library also includes an API for accessing client resources. Via this API, files can be uploaded, downloaded, or opened on the client. In order to accomplish such tasks, the Presentation Engine must be granted appropriate rights.

By means of an extension API, ULC can be enhanced to support new kinds of ULC objects, such as third party widgets or formatters. Adding new ULC objects is straightforward. Experience has shown that complex widgets, such as JavaBeans for business graphics, can be integrated with moderate effort.

For load and performance testing, there is a dedicated tool called ULC Load. ULC Load allows recording and playback of usage scenarios: series of user interactions can be recorded and replayed, simulating any number of clients, interacting with any number of servers. ULC Load is a valuable tool for determining server capacity and optimizing server configuration.

4.5 ULC Deployment

4.5.1 Client-Side Deployment

The Presentation Engine consists of two components, namely the engine launcher and the engine core. Both components are customizable and extensible. The launcher is responsible for creating the execution environment and for starting the engine core therein. For example, the launcher can be a standalone Java application creating a secure sandbox for the Presentation Engine. It can also be an applet running the Presentation Engine within a browser.

Java Web Start is a lean way both to deploy the Presentation Engine and to start ULC applications. By registering the standalone Presentation Engine as a helper application in the Web browser, a ULC based application can be started via a hyperlink in a Web page. Frequently used applications may also be accessed via a shortcut on the desktop.

4.5.2 Security

Similar to a Web browser, ULC offers a variety of options to meet stringent security requirements. In general, it is recommended to run the Presentation Engine in a sandbox, either as an applet or an application. Moreover, protocols like HTTPS or IIOPS can provide encryption and authentication.

In addition, the Presentation Engine can be adapted to different security scenarios. In a trusted environment it may run outside the sandbox if it is restricted to connect to trusted hosts only. The Presentation Engine can also be signed by a trusted entity and restricted to connect to specific hosts only. Finally, using the Java Security Architecture, restricted access to security sensitive functionality (e.g. access to portions of the file system) can be enabled.

4.5.3 Server-Side Deployment

A ULC based application may be deployed as a servlet in a servlet container or as a stateful session bean in an EJB container. The servlet container is the perfect choice for most deployment scenarios, especially for applications in the Internet.

An EJB container offers more sophisticated application management features, such as support for activation/passivation to achieve higher performance and scalability. For Intranet scenarios this may be a valuable alternative to Web container deployment.

The required protocol for the servlet container is HTTP(S). This is the standard deployment scenario for applications with users in the Internet since HTTP(S) is accepted by most firewalls.

4.5.4 Communication

The communication model of ULC assumes a unidirectional protocol and therefore fits perfectly into the J2EE world. In case of frequent and regular updates, ULC supports polling. While polling is not feasible for HTML-based applications, the incremental upload of changes in ULC does not lead to network contention.

Client-server communication is handled by two components: a connector on the client side and a container adapter on the server side. Out of the box ULC comes with connectors and adapters for the Web and EJB container. The actual protocol is determined by the J2EE container and can be configured at start-up time.

It is important to note that an application developer does not need to be concerned about communication components and the protocol. In the rare case where a new protocol cannot be configured, it is easy and straightforward to extend a connector and its corresponding container adapter.

5 Multi-Channel

Like elsewhere, the world of J2EE clients is not black and white. Often, applications need both a simple and a rich user interface. Some parts of an application may be well conceived and easily developed using HTML, while other parts require a rich user interface that is impossible or very expensive to develop with HTML. A possible solution for such mix & mach requirements is to use applets as an extension to an HTML-based user interface. Yet, from a development perspective, applets and HTML do not match well. Applets are client-centric, whereas an HTML application resides on the server. Thus, synchronization between the applet part and the HTML part is far from trivial.

In contrast, ULC can easily provide an additional channel to an HTML-based application. Both the ULC part and the HTML part can run within the same servlet engine. This allows for simple sharing of the session state, and of presentation as well as business logic. It is even possible to integrate such an application on the user interface level.

6 Conclusions

The Presentation Server approach substantially simplifies client-server partitioning, client configurations, and software distribution. ULC is a proven library for J2EE rich clients based on the Presentation Server approach. It successfully combines the advantages of an application independent thin client and the rich user interface of a Java-based application.

While HTML is sufficient for simple user interfaces, ULC is vastly superior as soon as sophisticated user interfaces are required. ULC also offers a homogeneous application development platform. In contrast, conventional servlets and Java Server Pages force the developer to deal with numerous different technologies including Java, HTML, DHTML, and possibly JavaScript. With ULC, the developer can focus on the design of the user interface and the application logic, and does not have to deal with distributed programming or bandwidth optimizations. In this way, development and operation of scalable, server-based J2EE applications becomes easy and cost effective.

7 Information

For more information on ULC see
http://www.canoo.com/ulc/.

Canoo Engineering AG
Kirschgartenstr. 5
4051 Basel
Switzerland
Tel +41 61 228 94 44
Fax +41 61 228 94 49
http://www.canoo.com/ulc/
ulc-info@canoo.com

Trademarks