ULC:Ultra Light Client Technology White Paper

Introduction

With the advent of the network and the Internet there have been many paradigm shifts in application development and deployment technologies. From the earliest days of the text based 3270 terminals to the more modern 2-tier and 3-tier client/server, graphical user interface (GUI) based applications. Each of the technologies has their strengths and weaknesses. The sheer simplicity of the server based 3270 applications has been weighed against the flexibility and the user-friendly nature of client/server GUI based applications. With the increasing popularity of the personal computer and the graphical user interfaced based operating systems, demand for feature rich applications has led application developers to build increasingly bloated client side applications that have been termed "Fat Client" applications. With the relatively recent introduction of web-based technologies like HTML and Java, developers have looked to these technologies to move to a "Thin Client" based computing model. Unfortunately HTML being primarily designed for page layout does not lend itself well to building complex applications. Java on the other hand has a robust development model and is well suited to building complex applications but does not address the issue of building thin client applications.

ULC a patent pending technology addresses these issues with a robust development environment based on Java, with a server based architecture well suited for building thin client applications. ULC allows the rapid development and deployment of lightweight, easily distributable, low bandwidth client applications.

RevaSoft ULC for Java is now available as an add-on to the award winning IBM VisualAge for Java development environment.

Executive Summary

The ***RevaSoft ULC for Java*** architecture consists of a thin Java based "Universal User Interface Engine" (UI Engine) and a ULC Server. The UI Engine allows all applications built using ULC technology to share the same client side code. The UI Engine communicates with server side ULC applications using a highly optimized ULC protocol that has been designed to operate on networks with speeds as low as a 28.8KB modem line. The ULC protocol is implemented over a ULC Transport abstraction that allows ULC applications to run using a variety of low level industry standard transports/protocols like TCP, HTTP, HTTPS, IIOP, SSL etc. In addition customers can implement their own ULC Transports to cater to any proprietary encryption or authentication requirements.

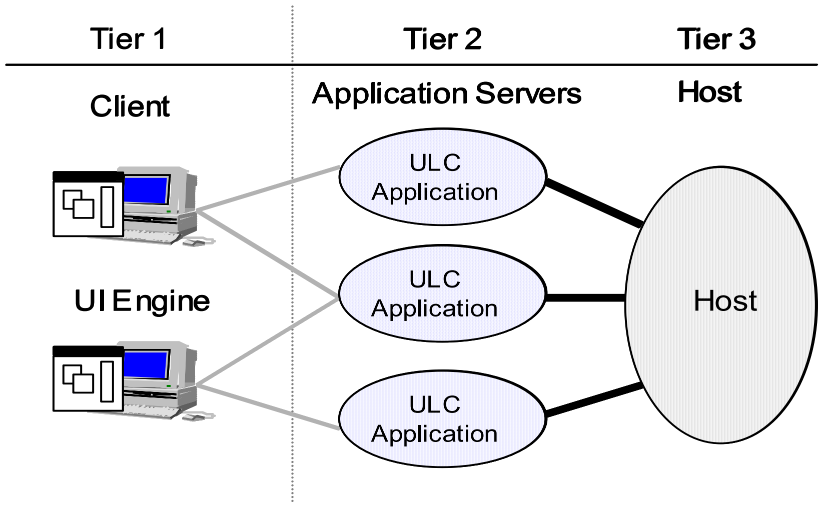


Figure 1: ULC Architecture

The UI Engine uses the Java-Swing widget library to allow function rich user interfaces to be built. The UI Engine is relatively small (~450KB), presents the views to the user, and performs local validation, formatting and editing of data without requiring round trips to the server. The ULC widget set which is loosely based on the Swing API allows developers familiar to Java Swing programming to easily switch to building applications using ULC.

A ULC Server application communicates with the UI Engine using ULC Proxy widgets that abstract all aspects of communication and distribution, allowing the developer to concentrate on building the application. All ULC applications transparently support lazy loading of data and widgets allowing ULC applications to use minimal amounts of bandwidth since features/data not explicitly viewed by a user will not be loaded into the client machine.

The ULC development environment gives the developer the flavor of building a single user fat client application while building a multi user server based application. ULC applications can be built visually using the Visual Composition Editor of the IBM VisualAge for Java development environment.

Since both the ULC Server and ULC UI Engine are built using industry standard Java they can be deployed to any platform that supports a Java2 compatible virtual machine. This currently includes a wide variety of platforms like Windows 95/98/ME/2000, Solaris, Linux, Mac, AIX etc. In addition since all ULC server applications do not have any Java-Swing user interface code, they can be deployed on Java platforms that do not currently support the Java Swing interface library (e.g. IBM's OS 390).

ULC applications are deployed on a central server allowing simplified administration, distribution and updates to clients worldwide. The UI Engine can be downloaded on the fly when the first ULC application is started.

Integration with the web is supported in many ways. All ULC applications can display web pages using the provided HTML component. In addition the UI Engine can be run as an applet within a web browser using the Java Plug-In, allowing all ULC applications to blend seamlessly with HTML based content. In contrast to building server based applications using HTML where often a developer has to build portions of the application using HTML, Java, and JavaScript etc. leading to incompatibilities depending on the web browser of the client machine, ULC provides a consistent programming model based on industry standard Java.

Thin Client Architectures

Moving to a thin client architecture has many advantages. At the same time there are different thin client architectures and choosing the right architecture for your organization can mean the difference between success and failure.
There are basically two popular thin client architectures available today namely "Display Servers" and "Presentation Servers".

Display Servers

Display servers have been around for a long time. The most widely used display server is the X-Window system deployed on most Unix based systems. In this model the entire application is run on the server, the client merely accepts input from the user and displays the output of the application. All input is passed directly to the server and is not interpreted on the client. The primary advantage of this architecture is that the client and client configuration is very simple. All the client has to do is support simple drawing primitives and convert all keyboard and mouse input into messages to the server. The primary disadvantage of this architecture is that since all keystrokes, mouse movements etc. are sent to the server the bandwidth usage of this architecture is relatively high and if the network in question is based on a wide area network, then network latency can play a very import part in the overall user experience.
More recently the Citrix WinFrame and the Microsoft Terminal Server products employ a similar display server approach to allow existing fat client applications to be deployed in a thin client environment. In an unlimited bandwidth scenario or in a local area network these products would

perform well but since most organizations have finite bandwidth availability to their clients other approaches have gained prominence.

Presentation Servers

Presentation servers are a more recent phenomenon. With the advent of intelligent terminals and higher CPU power on the client desktop it allows off-loading of a number of tasks from the server to the client. In this architecture the client performs all the display and presentation chores as well as handling most of the low level input handling. The client only passes higher-level messages/requests to the server. Keystrokes, mouse movement etc are all handled locally and hence greatly reduces the bandwidth requirements as well as the load on the server. The most popular presentation server available today is the ubiquitous web browser. The immense popularity of the World Wide Web has allowed the web browser to become the closest thing we have to a "Universal Client". Naturally the near universal availability of the web browser has caused developers to target the browser as an application development/deployment platform. Unfortunately HTML was primarily designed as a page layout language and not as an application development language. Though subsequent versions of the HTML specification have tried to address some of these shortcomings, HTML still remains an unsatisfactory language for building applications. Inconsistent implementations of the HTML specification in different web browsers further increase the developer's burden. Often developers have to build browser specific versions of each displayed page. HTML based widgets are very simplistic in their implementation and do not give the developer much control as to lazy loading of data etc. This makes it difficult to handle large volumes of data using HTML widgets since all data needs to be downloaded to the client before the widget can be displayed. In addition local validation and formatting is usually implemented using a script language like JavaScript increasing the complexity to the developer who now has to deal with yet another language.

ULC

The ULC architecture takes the basic idea of a presentation server and adds intelligent widgets, lazy loading of data and widgets, combined with a bandwidth-optimized protocol all implemented on a highly portable language environment like Java. The ULC architecture solves most of the shortcomings of the HTML based presentation servers and offers a homogenous Java development environment.

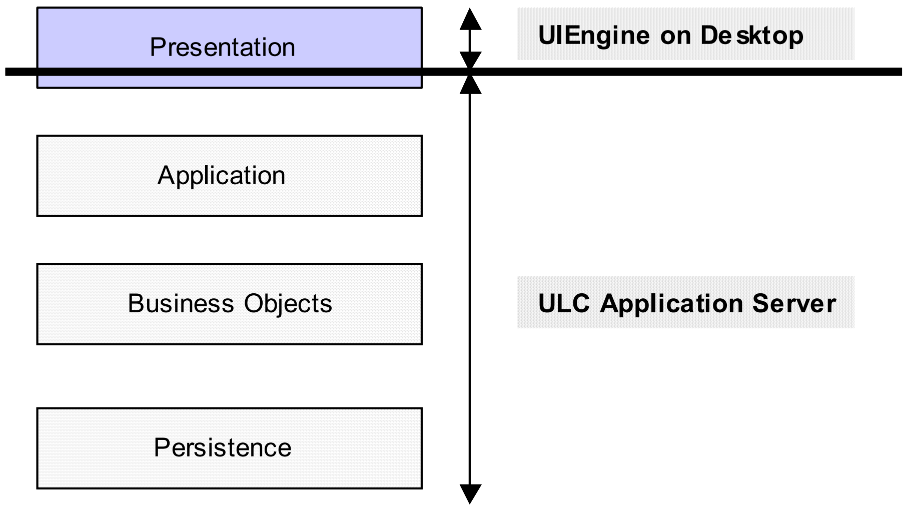
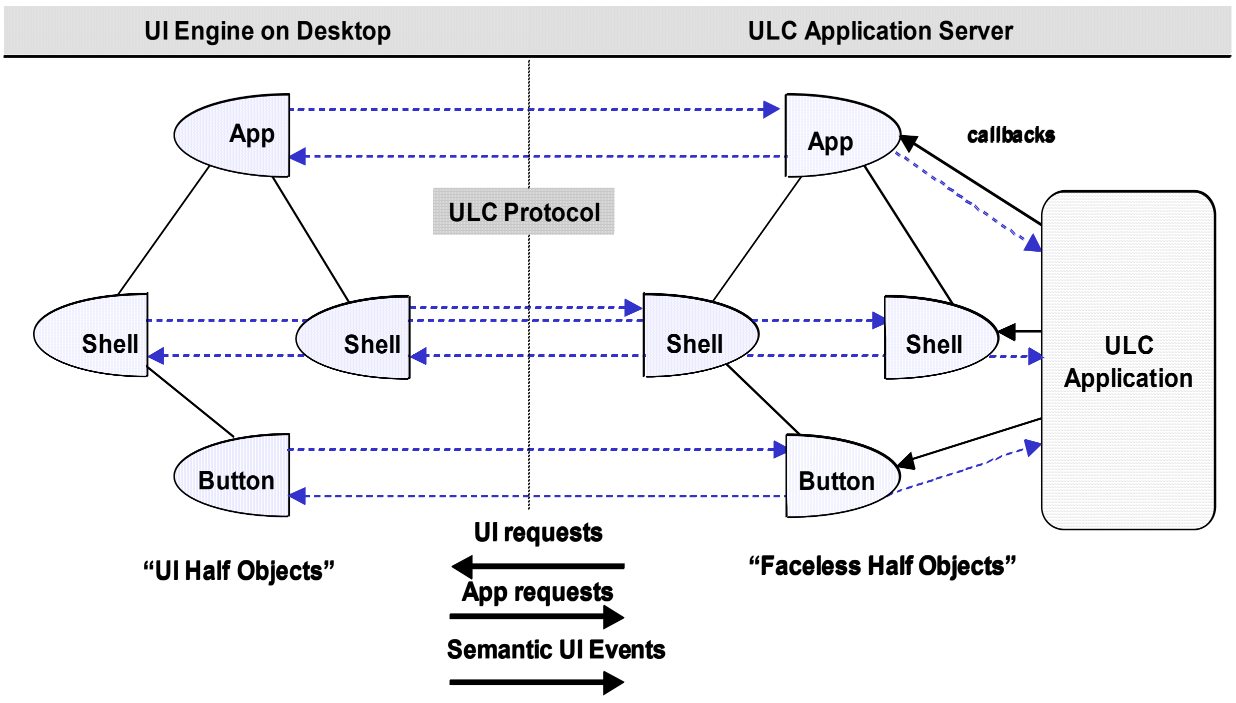


Figure 2: ULC Presentation split

Half Objects

One of the key aspects of the ULC architecture is that all ULC widgets are developed as two halves. One half of the widget resides on the UI Engine and is responsible for displaying the user interface and the other half resides on the server providing the application developer the API by which the widget can be manipulated.



<u>Figure 3: ULC Half objects</u>

Since each half is tailor made for its counterpart, communication between them can be highly optimized. Default information need not be sent across the network, as each widget's counterpart would automatically choose the same default values. The UI half of the widget also handles all low level user input like keystrokes and mouse events and only sends higher-level semantic UI events to the server. E.g. "button clicked", "entry field contents changed" etc. Events that usually have no significance to the application's functioning like "window resized", "mouse moved" etc are not sent to the server unless explicitly requested by the application developer.

Lazy Loading

All ULC applications automatically support lazy loading of data and widgets without requiring special effort from the application developer. Irrespective of the complexity of the user interface designed, the UI Engine will only request from the server those portions of the user interface that the user has actually navigated to. For example if an application developer built an user interface containing a notebook with many pages, only the first page will be uploaded to the UI when the view is first displayed. Additional pages will be requested on the fly only if the user navigates to those pages. In addition all data centric widgets like Lists, Tables, Trees etc will only request data that is currently visible. As the user scrolls down the list the UI Engine will transparently request from the server the additional data to be displayed.

Intelligent widgets

One of the primary aims of the ULC architecture is to reduce round trips between the server and the UI Engine. ULC widgets provide many services that significantly reduce the need for traffic between the server and the client. *Validators* and *Formatters* allow widgets to validate and display input data directly in the UI Engine. *Enablers* allow widgets to be enabled or disabled based on the state of another widget. For example a button could be enabled when a list has a selection or a button could be enabled when the user has modified a form that needs to be committed to the server. In addition Enablers can be concatenated together to allow any arbitrary combination of enablers to be linked together using simple Boolean logic. Widgets that accept user input have a configurable notification policy allowing the developer to choose when changes in the UI are committed to the server. For example when inputting data within a form, the developer could choose to change the notification policy to allow data in all input fields to be entered before the changes are committed to the server. Widgets that broadcast events to the server have been enhanced to support optional events. Events that have no listeners on the server side are not sent to the server further reducing network traffic.
Dynamic user interfaces are supported using a novel ULC Page book concept allowing multiple sub-forms to be automatically displayed by the UI Engine without requiring round trips to the server.

Asynchronous Communication

All ULC communication between the server and the UI engine is asynchronous in nature. This greatly enhances the scalability of both the ULC server as well as the UI engine. Since a single UI Engine can be simultaneously connected to multiple ULC servers, the user, while waiting for a reply from one ULC application can continue to work with another. This aspect allows ULC applications to stay responsive even in a wide area network where response times and network latency vary greatly. In contrast competing thin client environments like WinFrame and Terminal Server show distinct lags and pauses when used across a wide area network like the Internet.
Requests can optionally be batched together allowing multiple ULC requests to be sent within a single physical message send reducing the effect of network latency.

Layout and Widgets

ULC is designed to support a variety of client environments. Since the resolution and capabilities of the client machine cannot be known at development time, ULC uses a declarative layout mechanism to specify the user interface without having to specify actual widget sizes and positions.

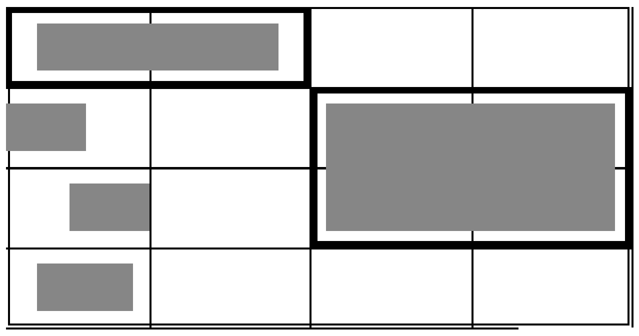


Figure 4: ULC Layout

The ULC layout mechanism is based on a simplified version of the Swing GridBag layout. The developer positions widgets within a virtual grid, specifying whether the widget can span across one or more cells of the grid, alignment within the cell is specified as well as the semantics of what should happen to the widget when its enclosing cell is expanded. This layout mechanism allows the UI Engine to optimally handle different screen resolutions and available fonts while offering sensible resize semantics.
The UI Engine supports a comprehensive list of widgets that are well suited to building commercial quality applications. For those organizations that require additional functionality the UI Engine can be

extended using custom widgets and third party Java beans that are then immediately available to all ULC applications.

Development Model

Developers familiar with building Swing based applications would find building ULC applications to be very easy. ULC keeps the same Model-View-Controller paradigm that Swing developers are familiar with as well as a similar event model. The ULC development environment integrates with IBM VisualAge for Java allowing visual construction of ULC applications thereby making building ULC applications extremely simple. Developers who prefer to write code by hand can bypass the GUI builder and write code by hand using the ULC API.

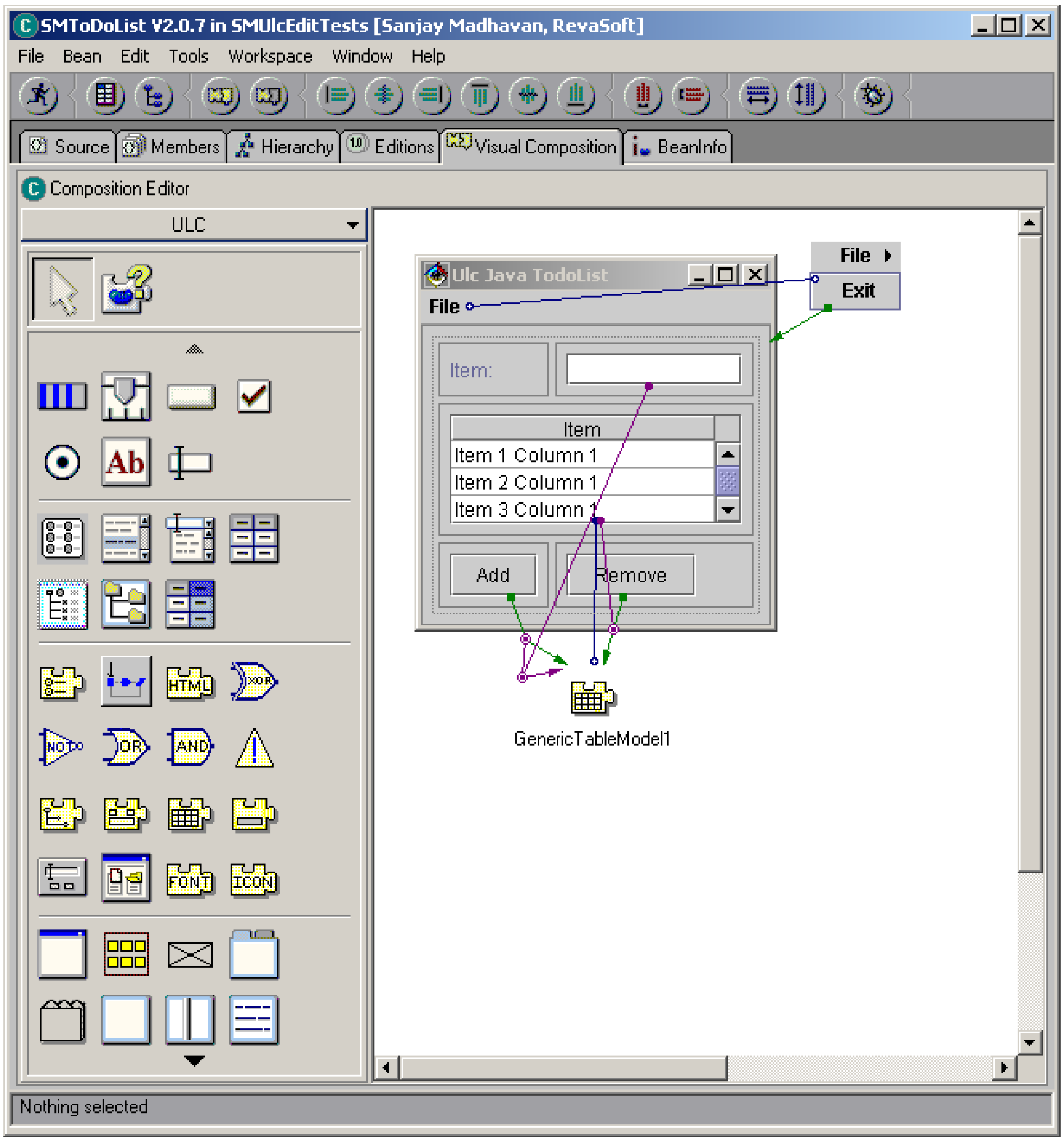


Figure 5: ULC GUI Builder

Deployment Model

Both the ULC UI Engine and ULC server allow multiple deployment scenarios.

Client-Side Deployment

The UI Engine can run as a standalone Java application or as an applet within a web browser using the Java Plug-In.
When running as an applet the UI Engine can be downloaded on demand to the client. When running as a standalone Java application one of the following approaches can be used to get the UI Engine to the client desktop depending on the organizations requirements.

a) Make the UI Engine available on a shared LAN drive.
b) Use an installer tool to deploy the UI Engine to the client on demand.
c) Use Java WebStart to automatically deploy the UI Engine the first time a ULC application is launched.

Of the above choices using Java WebStart is the preferred choice since the WebStart approach is a platform portable approach that Sun has announced will be included in the next release of Java.
In addition WebStart allows multiple versions of the UI Engine to be cached locally and has an extension and update model where fixes and user extensions can be deployed incrementally.

Server-Side Deployment

ULC server applications can run standalone using any Java compatible virtual machine. This is the simplest deployment scenario for ULC applications. This approach is well suited for installations with a moderate number of users. For installations with a large number of users load balancing issues need to be addressed by integrating ULC applications into the organizations existing load-balancing infrastructure.
ULC server applications can also run as servlets within any Servlet 2.x compatible servlet engine. This approach leverages existing web server infrastructure and potentially resolves load balancing issues as well since most organizations already have load balancing infrastructure for web based servlets in place. Running ULC applications as servlets also demonstrates the capability of the ULC HTTP Tunneling transport that is key to penetrating client side firewalls.

Summary

While HTML is suited for simple applications, ULC is well suited for building complex applications. ULC provides a consistent Java based development platform allowing the developer to concentrate on application functionality while the ULC framework handles distribution and communication issues. The ULC development environment and the integration with the Visual Composition editor of IBM's VisualAge for Java provides the developer with a powerful tool for rapidly building and deploying thin client application.

ULC brings the advantages of thin client computing to Java.

Trademarks

Citrix and WinFrame are either trademarks or registered trademarks of Citrix Systems, Inc.
IBM and VisualAge are either trademarks or registered trademarks of IBM Corporation.
Windows and Windows Terminal Server are either trademarks or registered trademarks of Microsoft Corporation.
Java, Java Plug-In and Java WebStart is a trademark of Sun Microsystems, Inc.
X Window System is a trademark of X Consortium, Inc.
All other tradenames referenced herein are trademarks or registered trademarks of their respective holders