

# Never Too Rich or Too Thin

by Bernhard Wagner

## Following the middle road

**Y**ou can never be too rich or too thin. That's what Wally Simpson might have quipped to her stock trading application had she lived to enjoy the blessings of the Internet. Indeed, Wally may have had a point there: today's mainstream approaches to end-user computing are lacking. Fat clients are difficult to distribute and HTML is inadequate for high-end GUIs. A client that is both rich and thin would be the ideal solution.

### Rich Thin Clients

Java is a great platform for rich thin clients (RTCs). The availability of JREs on both the client and server is a formidable basis for an RTC library. Given standard Java infrastructure, such a library can offer server-side peer objects for widgets and a presentation engine executing the GUI for any number of applications, as shown in Figure 1.

This library will be lean and mean because it can:

- Delegate event handling and graphics functions to the JRE on the client
- Profit from the fact that JRE runs both as a plugin within a browser and on the desktop
- Make use of Java Web Start for distribution
- Leverage J2EE for communication and server-side modeling of the user interface



Bernhard Wagner is an independent software consultant and longtime developer of rich-client software. He developed a visual programming environment allowing the visual composition of Multimedia components, 3D direct manipulation applications, and numerous HTML applications.

bw@xmizer.biz

An RTC library is not a panacea. Yet it may go a long way toward rich GUIs without getting chubby or sacrificing the core advantages of HTML. As we'll see later, a number of products testify to this point today.

Let me discuss some key characteristics of a well-designed RTC library and how it can profit from Java.

### Never Too Rich

From the perspective of usability, an RTC presentation engine should support as many rich client functions as possible. The functionality must be

significantly better than HTML's, offering:

- A more responsive interface that minimizes server round-trips
- Direct manipulation
- Comfortable widgets like tables with self-sorting columns, trees, and high-end editors
- Superior integration of desktop functions

### Never Too Thin

From the point of view of distribution and operation, an RTC presentation engine should be as lean as possible, that is:

- Free of application-dependent code so that the rollout of applications is server-side only
- Sufficiently small to enable distribution as an applet
- Use existing J2SE and/or J2ME infrastructure wherever possible

Evidently there is a trade off between the wish lists for thin and rich. This suggests a further requirement: an RTC engine must be as lean as possible in its basic form, but extensible. It should come as a slim core library with plug-gable extensions and an API allowing the integration of existing rich client libraries.

### Optimized Communication

An important bonus of Java-based RTCs is that their network bandwidth

requirements can be minimized. Typically communication will be several times more efficient than for HTML for the following reasons.

First, server round-trips can be slashed by executing tasks within the presentation engine, for example:

- The enabling and disabling of widgets that depend on each other
- Syntactic validations and formatting of text fields
- Sorting of lists or columns
- Caching

A second means to minimize communication is to model the status of user interfaces on the server. This enables the session to keep track of what is visible on the client and thus limit data transfer to visible items.

### Server-Side Programming Model

Designing a client/server application is substantially simplified when a server-side programming model is employed. This avoids the difficult issue of splitting functionality between client and server.

A well-designed RTC library enforces a server-side model. Notice that this will exclude an approach in which developers specify code that is transferred to the client and executed there. This latter approach leads to fat client programming, which is undesirable.

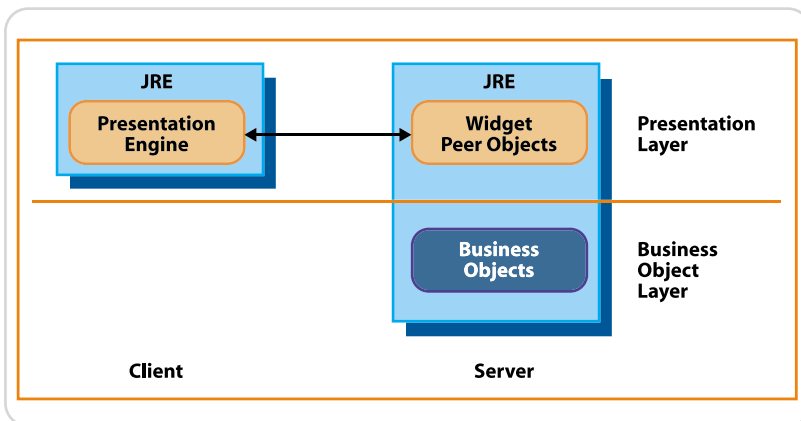


Figure 1 JRE-based architecture for rich thin clients

## Seamless Object-Oriented API

A further feature for an RTC library is a seamless object-oriented design and API. Ideally, an RTC library offers a server-side API that corresponds to the API of a well-known library like Swing, AWT, or SWT.

The benefit of such a design is that a cumbersome mix of technologies can be avoided: instead of cobbling the GUI together with Java, JSP, HTML, or proprietary XML languages, the developer can use a seamless Java API.

## Server-Side Execution

As mentioned, downloading user-defined code to the client for execution is undesirable. An RTC system must execute everything on the server, except for the GUI.

In fact, even the model of the GUI must reside on the server in order to optimize communication (see above).

Notice that server-side execution is also an advantage for security as it can be handled much easier on the server.

## Pluggable Communication

The communication protocol is one of the major issues for client/server applications. Depending on the environment in which an application must run, different protocols must be used.

For this reason, an RTC library should isolate client/server communication in pluggable modules, such that an application can be configured to run over HTTP, HTTPS, RMI/IIOP, or other protocols.

## Standalone/Offline Execution

With a Java VM on both the client

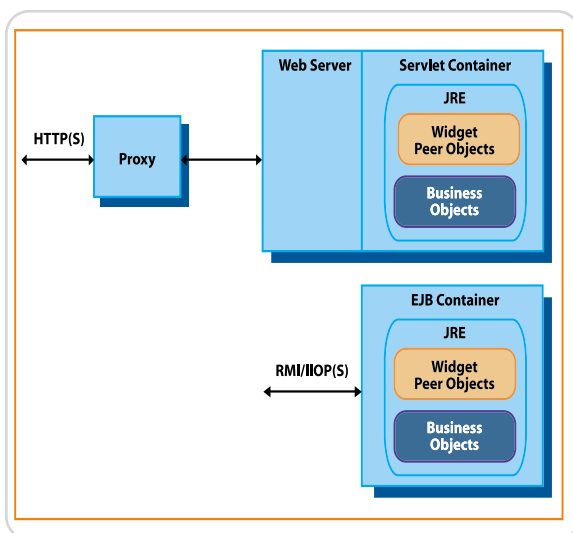


Figure 2 RTC library leveraging J2EE

and server, an RTC library can enable flexible on/offline execution with a single code base.

If pluggable communication is in place, all that's needed is a module that simulates client/server interaction. Such a module will allow you to run a client and server in a single VM and thus on a standalone machine.

Applied cleverly, standalone execution will serve three purposes. It will:

1. Simplify development because the edit/compile/test cycle can be executed locally and within the IDE
2. Support incremental growth – applications starting as a single user can be converted to multiuser with minimal effort
3. Enable development of applications that can run both online and offline

## Leveraging Standards

J2EE, J2SE, and J2ME provide a superb standardized platform for RTC systems. All basic functions required for the GUI, communication, server sessions, and security are readily available. As a consequence, a well-designed RTC library is just a thin software layer that essentially provides a server-side API for the standard widgets of Swing or AWT, delegating everything except the core RTC functions to the standard libraries. An API for SWT is, of course, also an option for cases in which the client-side presentation engine relies on SWT. Figure 2 shows how an RTC library can be embedded into a standard infrastructure.

## Leveraging Existing Infrastructure

Focusing on core RTC functions is a key requirement, not only with respect to standards. An RTC library should be designed to integrate with the existing infrastructure. It should, for example, fit into an existing platform for HTML applications, enabling a mix and match of HTML clients and rich clients, as well as multichannel applications that share everything up to the presentation layer.

An RTC library is, therefore, typically an extension of an existing software platform and not a platform of its own.

## Products on the Market

A number of products illustrate how it can be done: AltioLive, AppProjector, Canoo ULC, Classic Blend, Droplets, RSWT, and Thinlets.

All of these are Java-based RTC libraries. They have put a different emphasis on the defined requirements, and some of them are not pure Java but are hybrid, employing proprietary XML languages. Their common denominator is that they all prove the viability and usefulness of Java for RTCs. Products that forego the advantages of JRE on the client are available as well: examples are Classic Blend and Macromedia Flex. They use JavaScript and a proprietary execution environment on the client, respectively.

## Conclusion

Today's mainstream approaches of the HTML thin client and the fat client productivity application are antipodes. The one's strength is the other's weakness.

The rich thin client (RTC) is the middle road that often succeeds in offering the benefits and avoiding the weaknesses of both. Such magic is not possible for all scenarios, but for many client/server applications.

Various Java-based RTC technologies exist today. Java is particularly suitable for RTC libraries because of its cross-platform availability and broad standard infrastructure. Most important, Java enables a seamlessly object-oriented, server-side programming model that avoids the cumbersome mix of technologies with proprietary XML languages, JSP, HTML, and others.

Now that the rich client is becoming popular again, we may expect that many of those who have experienced the benefits of HTML will not be happy to cope with fat clients again, or with the prospect of building a new infrastructure for client/server computing from scratch. Some of them may go with Ms. Simpson's advice and choose Java RTCs. ☺

## References

- *AltioLive*: [www.altio.com](http://www.altio.com)
- *AppProjector*: [www.asperon.com](http://www.asperon.com)
- *Canoo ULC*: [www.canoo.com/ulc](http://www.canoo.com/ulc)
- *Classic Blend*: [www.appliedreasoning.com/products\\_what\\_is\\_Classic\\_Blend.htm](http://www.appliedreasoning.com/products_what_is_Classic_Blend.htm)
- *Droplets*: [www.droplets.com](http://www.droplets.com)
- *RSWT*: <http://rswt.sourceforge.net>
- *Thinlets*: [www.thinlet.com](http://www.thinlet.com)
- *Classic Blend*: [www.appliedreasoning.com/products\\_what\\_is\\_Classic\\_Blend.htm](http://www.appliedreasoning.com/products_what_is_Classic_Blend.htm)
- *Macromedia Flex*: [www.macromedia.com/software/flex](http://www.macromedia.com/software/flex)