

Jiffybus Background Whitepaper

By Scott Came, ANZUS Technology Inc.

First, a bit of history

In the early 1990s, desktop workstation computers became powerful and inexpensive enough to warrant wide deployment in the enterprise. Gradually, enterprise applications once conceivable only on the mainframe platform were being ported to and deployed on PCs. As installations of Microsoft's Windows operating system became more widespread, demand for Windows-based user interfaces grew, and so-called "client-server" tools entered the market to build these user interfaces. In general, application users were delighted with the aesthetic and usability improvements of these applications versus the "green screen" applications they had been using on the mainframe platform.

However, adoption of the client-server architecture was slow in the enterprise, especially for large-scale, corporate-wide applications. While client-server applications could easily be deployed to a small, local workgroup, scaling them to thousands of geographically-dispersed users proved difficult and costly for IT departments. The traditional client-server development strategy was to partition application functionality into two segments: the data store or database, and the client application. The database generally resided on a server machine, and was accessed by the client over a local- or (less successfully) wide-area network. All application code--both business logic and user interface logic--was bundled into the client application, which had to be installed onto each desktop workstation. IT faced the problem of visiting desktops anytime a change was made or a bug was fixed, which proved to be terribly costly. In addition, a constant cycle of desktop hardware upgrades was started to keep the processing power in step with the sophistication of the applications being developed.

In the late 1990s, as key internet technologies improved, enterprise IT began recognizing the potential of the Web as an application deployment medium. It solved virtually all of the client-server deployment and scalability problems. The web browser became a ubiquitous container in which content could be deployed easily and securely. Changes to application functionality needed only to be redeployed on one machine: the web server. And the fact that web browsers could run fine on slower hardware lengthened the hardware upgrade cycle and freed IT resources for application development. And, as an added bonus, the main internet technologies--HTML, browser scripting languages like JavaScript, and server-side scripting technologies like Microsoft Active Server Pages (ASP) and Java Server Pages (JSP)--significantly shortened the learning curve for IT developers, compared to the sophisticated client-server development tools of the mid 1990s.

The unfortunate downside to the browser as a client-side deployment vehicle was that much of the improvement in user interfaces brought about by the client-server revolution was lost. Computer-users soon found that HTML and browser scripting languages place severe restrictions on the richness of user interfaces that can be built. In large part, this is

because HTML started out as a technology for displaying rich documents on the web, not as a mechanism for building applications meant to interact with a user. That is, HTML was designed to be *read*, not *used*. Certainly, the World Wide Web Consortium and the browser vendors have made strides forward in enhancing HTML with user-interface capabilities. And for many simple applications, the limited user interface capabilities of the HTML+scripting solution can easily handle the business requirements. But for a large share of enterprise applications, where a user must interactively view, edit, manipulate, and visualize large amounts of data, HTML has proven woefully inadequate. In addition, despite the existence of standards for most of the components of this architecture, each popular web browser supported its own set of capabilities, especially in the area of advanced user-interface support. The cross-browser, cross-platform promise of the web has never fully materialized.

Coincident with the rise of the Internet, a team at Sun Microsystems developed an application development platform called Java. Java is a language, API, and virtual machine architecture that permits deployment of cross-platform, network-aware, multithreaded, object-oriented, secure applications over the Web. Soon after its release in 1995, Java applets--applications delivered over the web via HTTP that run within the browser context--began appearing on the internet. There was initially some hope that applets could bridge the gap between the scalability and ubiquity of the web application paradigm and the richness and usability of the client-server paradigm. However, adoption of client-side Java was very slow in the enterprise. In addition to a much steeper learning curve than HTML+scripting technologies (and even steeper than the client-server development environments of the early 1990s), Java out-of-the-box suffered some significant architectural flaws.

Applets were initially conceived as an extension of the web browser. However, the core Java API offered very little assistance or guidance on how to do what browsers do best: communicate with a web server over HTTP, formulating HTTP requests and listening for HTTP responses, and customizing those requests and responses to transmit structured enterprise data between application tiers. While the core Java API did provide Remote Method Invocation (RMI) as a means of communicating between applets and a server, the protocol itself had scalability problems and suffered from the fact that it simply was not HTTP. And though the API did have some functionality for dealing with the HTTP protocol, it wasn't enough to inspire enterprise use of applets as a core technology.

The Jiffybus Solution

The Jiffybus toolkit is an API and framework designed to allow low-cost deployment of n-tier applications with rich, client-server style user interfaces over wide-area networks, such as the public Internet. As this is a mouthful, let's examine each of these characteristics in detail.

- **...an API and framework...**

Jiffybus is not an application server or other standalone middle-tier solution. Rather, it is an API and set of relatively simple tools that web developers can use to deliver solutions more easily and robustly. It assists the developer by facilitating communication between the tiers of a multitiered application, and by providing a basic architecture to help partition application behavior appropriately.

- **...designed to allow low-cost deployment...**

Jiffybus takes advantage of [Java WebStart](#) technology from Javasoft to deploy application code automatically to the client on demand using HTTP. (Jiffybus also allows applications to be deployed as applets hosted within the Java Plugin in the web browser, but WebStart is becoming the preferred industry standard for client-side Java applications.) All client-side code is deployed onto the web server, and is delivered to the user's browser automatically the next time he or she starts the application. Consequently, IT reaps the deployment benefits of the web platform, while giving users a rich, client-server style user interface.

- **...of n-tier applications...**

Jiffybus provides an API and supporting infrastructure for developing business services separately from presentation logic. Of course, Jiffybus does not guarantee that applications will be architected properly. Rather, it gives the developer the tools needed to implement an n-tier architecture easily, with a Java WebStart client as the front-end. Again, IT keeps the high scalability inherent in web applications, but preserves as well the richness of the client-server style user interface.

- **...rich, client-server style user interfaces...**

Jiffybus includes supporting infrastructure to render structured enterprise data in Java Swing components. Swing is the next-generation cross-platform UI toolkit that is, as of Java 2, part of the core Java API. It includes most of the widgets and other features developers expect to find in a client-server style UI toolkit. Where Swing is lacking, Jiffybus contains (in some cases) Swing extensions to provide the missing functionality. In many cases, data structures created on the middle tier can be bound to UI components, which will display the data with a minimum of custom programming.

- **...over wide-area networks...**

Jiffybus uses HTTP to communicate between the client and middle tier. Consequently, this communication can occur across any network architecture that supports HTTP, including the public Internet. It easily passes through firewalls and scales as well as any HTTP-based application.

The bottom line is, Jiffybus provides a way to have the best of both worlds. On the one hand, it preserves the inherent scalability and low-cost deployment of the web application platform. In this way, it satisfies the requirements of IT shops that seek inexpensive, low-risk, industry-standard ways of deploying business functionality across the enterprise. On

the other hand, it satisfies the user by introducing dramatic improvements in user interface quality and functionality--things users grew to expect during the client-server heyday of the mid 1990s.

Now that you have the background, continue with these links to explore further the capabilities of Jiffybus: [Technical Whitepaper](#) [Frequently Asked Questions](#)