BBj's CUI Gets a Big Performance Boost

Our bodies are extremely efficient at processing the food we eat so that our internal workings run proficiently. If you are like me, when I work around the house like cleaning the gutters and mowing the lawn, I always look for the fastest and easiest way to accomplish those tasks. After all, the faster I can get a task done, the more time I have for more pleasurable activities! I might even look for more creative ways to perform those tasks and take advantage of new technologies. For example, instead of walking to work a few miles away, I might drive a car or take public transportation, or even work "virtually" by way of the Internet.

So why would BASIS do anything different with its products?

The Challenges

In prior versions of BBj®, we used an API called "Remote Messages" that passed information between the server and client for character user interface (CUI) displays. Originally, these remote messages passed character data from the server to the client in the form of Java strings stored as bytes. As BBj began to grow in functionality and expand into other world markets, we discovered that an array of bytes was not the same as a string that contained character values with those bytes. This problem first surfaced when our European customers used the Euro character (€). The Euro character displayed differently depending on the character encoding in use, e.g., ISO-8859-15 vs. UTF-8, thus changing what the user saw depending on their configuration. This means that the client and the server must use the same character encoding in order for characters to display correctly on the client and to ensure that characters typed on the client transfer faithfully to the server.

By Aaron Wantuck
Software Engineer

By Adam Hawthorne
Software Engineer

Each Remote Message also required an explicit class to store the data. In some cases a Remote Message required an additional response class, doubling the amount of code in BBj. Correctly creating a new Remote Message meant jumping through many hoops of implementing specific interfaces and methods. This excessive "boilerplate" code required to even implement a single message became difficult to maintain and use. It was too clunky, which lead to further compromises and inconsistencies within the code as the years passed and development progressed.

As we created more messages to handle increasing functionality, the data sent between the client and server grew. This led to communication problems as our customers began to switch to higher latency mobile networks and distributed systems. In a market where the need to stay competitive is paramount, this excessive communication over a high latency network could have detrimental effects on our future products becoming too slow and impractical for distributed mainstream business or personal use.

The compromises inherent in the original Remote Message API led to a poor abstraction in the CUI subsystem between channels and devices. Developers expect an OPEN on two distinct aliases to produce two distinct SysWindow devices. Applications can also open the same device on different channels. Each channel has its own isolated state and operations. To address this issue in the original implementation of CUI, the client maintained a map of CUI instances that resulted in unnecessary sharing between devices on the client. It was not feasible to put this information into the server because of the limitations of the Remote Message infrastructure.

Furthermore, since changing the contents of a Remote Message became such a daunting task, it became necessary for the server to provide the client with an up-to-date view of its server-side environment. Since the source of the state was decoupled from the consumer of that state, the server simply sent all the necessary data instead of only what had changed or what was absolutely necessary. This added to the network overhead as any change to the environment required the server to send another large message to the client so its view of the server-side state would remain consistent.

These issues cried out for a more efficient solution.

The Solution

BASIS experienced similar problems in SYSGUI, the graphical user interface, several years ago. At the time, the engineering team designed a generic library modeled after a Java technology called RMI (Remote Method Invocation). RMI enables developers to write object-oriented code so that objects on different computers can interact in a distributed network. However, the Java RMI design requires a response for every remote method call, which significantly increases the latency of every operation, especially on a slow network. BASIS adapted the architecture of Java RMI to overcome these inherent performance penalties by relaxing restrictions so remote methods may execute without requiring a response. RMI then sends these asynchronous messages in batches, further reducing the network overhead. The library automates as many of these optimizations as possible to reduce the burden on developers.

This simple design of automatically mapping a single method call to a message eliminates the need for extra code and extra objects to implement a single message. Adding a new message is as simple as adding another method to a pre-existing RMI interface and then providing the implementation in the remote class. A developer needs only to consider what operations are actually necessary, and can spend more time perfecting the implementation than on producing the infrastructure.

In CUI, since the individual channels are instances on the server and two channels can easily communicate with the same client device, the client does not have to maintain a “map of instances” any more. If the client requires any information, the server can preemptively send the data to the client just by adding another parameter to a remote method. This completely eliminates the need for the server to continually refresh the client's view of its state. If a particular message requires a particular piece of server state, the server simply sends that single piece of information as a new method parameter. It reestablishes a proper relationship where the server now handles all the device and channel information.

The Results

As a result of all these changes, the user can enjoy faster response times since there is a drastic reduction of network traffic, and BASIS can enjoy cleaner and easier code maintenance. In fact, everyone can enjoy the benefits of this win/win situation.

Neither the server nor the client transfer character data in the native platform encoding. All character data is sent as standard Java String objects, using the ubiquitous UTF-8 character encoding. It is no longer necessary to configure the server and client to have the same character encoding, and both the server and client require less work to translate String objects back and forth into the native platform encoding.

The graph in **Figure 1** shows the results of the new implementation of RMI (red) and the old implementation of Remote Messages (blue). The tests ran a total of fifty times; five sets of five each for RMI and Remote Messages. Each set contained an average time for its set, providing five points for both RMI and Remote Messages. The resultant data was then plotted on the graph, clearly showing marked improvement in the new code. Note that the RMI time never exceeds that of the old remote message time.

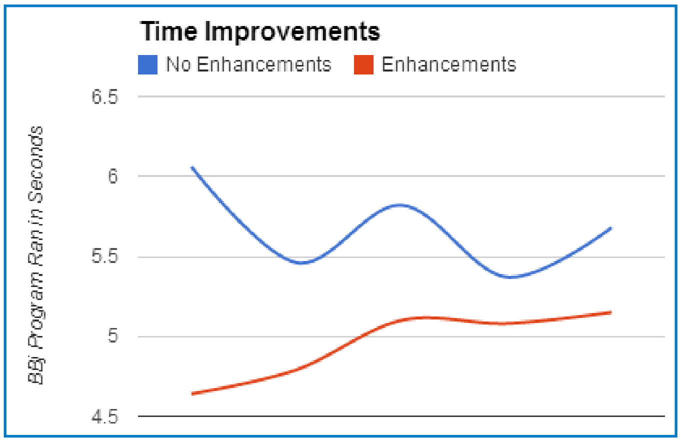

Figure 1. The new implementation of RMI (red) and the old implementation of Remote Messages (blue) in a high latency environment (USA-Europe)

Users can clearly see these enhancements. If they use the SysConsole with a Web Start program or the TermConsole to start any character based application in BBj, they experience an average improvement of around 25% and a much more consistent interface.

Summary

Enhancing product performance is an ongoing activity at BASIS, along with analyzing new ideas for further improvements. Using these future concepts, we can further decrease lag, add more features, and increase the responsiveness of our products, allowing more time for other tasks to run unnoticed. If you think of code as a living entity that is constantly adapting to new stimulus, there is so much more to learn and improve over time. We’ve seen tremendous adaptations in the medical field, greatly improving from the dark ages to today with such modern discoveries as previously unheard of organ transplants and artificial hearts, to name a few. With software still in its infancy, at just a few decades old, just think what we can do to improve it in another 10 or 20 years! ■

Learn more from these reference materials:
- *The Java EE 5 Tutorial* at links.basis.com/javaee5tutorial
- *Java RMI Tutorial* at links.basis.com/javarmitutorial