Loading with Class

By Chris Hardekopf and Jerry Karasz

Classloader Overview

BBj's loading of Java classes became more flexible and powerful with the release of BBj® 19.10. Loading Java classes for a client program is also more efficient than ever before.

But before we start digging into how loading classes has changed, let's take a moment and talk about what classpaths and classloaders are.

What is a Classpath?

A classpath is essentially an ordered collection of places for the Java interpreter to search when it needs to find the definition of a class. Usually, it's an ordered list of Java ARchive (JAR) files, each of which contains one or more Java class definition files. For more information on classpaths in BBj from version 19.10, please see the Advantage article *Getting on the Right (Class)Path*.

Please remember: Classpaths are only for finding *Java* class definitions so that they can be loaded by a classloader.

What is a Classloader?

Once Java searches the appropriate classpath(s) and finds the class that the code requests, it loads that class information into the Java Interpreter so that it can be executed. This is done by a Java classloader.

The New Classloader

In BBj 19.10 and higher, BBj uses a new classloader to load the Java classes it needs. For both Desktop App (remote deployment) and ThinClient executions, the loading process is the same: a client BBj program contacts the BBj server, asks it for the classes it needs in order to run, and then downloads those classes from the server. This on-demand class loading varies from the previous model in a number of ways.

Because the Desktop App solution actually uses the standard BBj ThinClient program execution, the rest of this article will only talk about ThinClient programs. But remember: everything in this article also applies to Desktop Apps.

Classes, Not JARs

The new classloader only delivers individual classes, not entire JAR files, to the clients. In other words, if a program needs only a portion of the classes stored in a single JAR file, only that portion needs to be downloaded. These savings can be significant, especially in a bandwidth-limited environment.

But What About Third-Party JAR Files?

If third-party JAR files are needed on the client, then they must be included in a Session Specific Classpath (SSCP), and the Thin Client must be configured to use that SSCP.

But What About Native OS Libraries?

Prior to BBj 19.10, native libraries were available to the client via a `System.loadLibrary()` call. From 19.10 onwards, there are now two options:

1. On the server: extract the native libraries into the appropriate `<bbjhome>/libnative` directory. For example: `/usr/local/bbj/libnative/windows/amd64/64`

   ```
   [root@server1 64]# ls
   b3ddePlugin.dll    NativeLicense.dll  RegPlugin.dll   User.dll
   NativeInstall.dll  NativeUtil.dll     rxtxSerial.dll
   ```

   Using `System.loadlibrary()` will then automatically deliver the requested library to the client.

2. On the client: place the native libraries in a JAR file and include a reference to the JAR file in the program's SSCP. The benefit of using this method is that it is self-contained and will work with any type of client. Of course, the client Java code will need to extract the native library resource before actually loading the native library.

What are the Benefits of the New Classloader Model?

The new classloader includes a number of optimizations to minimize the amount of information that must be transferred from the server to the client, thus increasing its performance. These optimizations are described below.

With Web Start, the JNLP file had to contain a full list of the JAR file dependencies for the remote deployment. Each JAR file would be downloaded in its entirety before the program started up for the first time (often making the initial startup very slow, and requiring a potentially significant understanding of the JAR file interdependencies).

With the new classloader model, each client program requests the classes it needs as it finds that it needs them. There is no requirement to explicitly pre-define all of the JAR files or the classes that may be needed at run time. The client will request them as they are needed, and the server will search its Java classpath, find them, and make them available to the client.

Client Caching

The client classloader first looks locally (in a client cache) for a class it needs before requesting it from the server. This local cache is keyed by version information from the server it is talking to. Based upon what it finds, it does one of the following:

- If it does not find a local version, it requests the class from the server, then adds the class to its local cache once it is received.
- If it finds a local version, it can then use it safely because of the version key.

This client caching minimizes the amount of class information that must be requested from the server, thus providing improved response times.

Server Caching

When the server receives a request from a client for a particular class, it uses its Java classpath to find the requested class information and load it into memory. It also checks its cache for information about what other classes have been requested in the past soon after this class was requested. Based upon what it finds, it does one of the following:

- If it finds none, then it simply sends along the requested class.
- If it finds any, then it also loads those predicted classes, and transmits the larger bundle to the client. This predictive download model minimizes the number of round-trips between the client and server, and maximizes the likelihood that the client will actually need the provided extra classes.

This server caching has a large impact on client programs, because it can help more than just the one client program's performance. Once the server cache contains information about the full set of classes that one client needs, then it can predict that other clients will also need those classes, and do a predictive download.

Because of this, one way to speed up all of the clients' startup times is to do a test run of a ThinClient application as part of setting up the application for the clients. Once this test run has been completed, every client that runs that application afterwards can benefit from the server's predictive cache information (because of the minimized number of round trips involved in the downloading).

Server-Client Version Synchronization

With the new classloader model, the ThinClient program will use the same version of each class that the server uses. This also means that all that a client computer needs is a current Java installation, along with one BASIS JAR file: `BBjStartup.jar`. The `BBjStartup.jar` contains just enough Java class information to help the ThinClient find and connect to the appropriate BBj server, and to begin

the class downloading process. It is guaranteed to not have any external dependencies on BBj classes or third-party JARs – it only needs Java.

For a Desktop App remote deployment, the BBj ThinClient will also check that it is using the correct version of Java (as defined on the server for this Desktop App), and if not, it will retrieve the correct JRE (as defined in the Enterprise Manager) from the BBjServer. The ThinClient program will then download all of the Java classes that it needs from the server as it needs them, thereby guaranteeing that the two are always using the same versions.

For more information on configuring and using Desktop Apps, see the "*Desktop Apps, a Zero Deployment Strategy*" Advantage article, coming soon!

How to Get Started?

So how does someone get started using the new classloader with its optimizations? There is no action required – anyone who starts a BBj program as a ThinClient or a Desktop App will automatically get all of the benefits of the new classloader. No special provisions or changes to code are required beyond using BBj 19.10 or higher.

Summary

With BBj's new classloader model, class information is loaded one class at a time – BBj no longer needs to download an entire JAR file. Class downloading is optimized by having a cache on the server side, and another cache on the client side. Together with the predictive class algorithm, these caches help minimize the number of round trips between the server and client, which increases application performance – particularly on low bandwidth networks.

The new classloader model is just one piece of the solution to replace Web Start with BASIS's Desktop Apps solution, and it is mostly unseen by users. But good performance should never go unnoticed, so stay tuned for the next article in this series, "*Desktop Apps, a Zero Deployment Strategy*", to learn about improvements that users are sure to see with their applications!