What I Learned This Month:

Java on the Mainframe: Untapped potential?

Scott Chapman
American Electric Power

As I've been experimenting with some different Java work on the mainframe recently, I've begun to think the mainframe's Java serving capabilities are underappreciated. That's probably because it wasn't too long ago that Java performance on the mainframe wasn't exactly ideal. Some would probably call it other names, but in my experience it was generally "ok" if you looked at what performance you *really* needed. But it certainly wasn't efficient and at times it was downright embarrassing. I wouldn't call it embarrassing any more. Processors running at over 4 GHz and specific hardware instructions to support Java certainly help!

From a software perspective things have gotten a lot better, too. JVM startup time has been dramatically improved; I can now run a simple "hello world" application in less than a second. Trivial, yes, but it wasn't always like that. There's a good range of tools built into the current JVM too: multiple garbage collection options, a fair bit of debugging data, and a profiler. At least some of those are specific to the IBM JVM, but the "standard" JVM options and features have also increased over the years.

One of those (that has been around for a while but I only recently learned about) is "Dfile.encoding". On the mainframe, add "–Dfile.encoding=ISO8859-1" to your run-time options and you're effectively running in ASCII mode, not EBCDIC! This means you can probably pick up and run almost any Java application on the mainframe as-is. I've successfully run the open source pure Java H2[1] database on the mainframe. I've also been playing around with Helma[2], a JavaScript application server that is written in Java. In both cases, I simply uploaded the class files and ran them in a JVM with the above runtime option. You've got to like it when porting an application to a new platform only involves FTP!

Helma uses Mozilla's Rhino as its JavaScript engine. Interestingly, new in Java 6 (although Java 6 isn't really new) is the ability to interpret script code via a script engine. The standard Java 6 engine uses Rhino to interpret JavaScript, and yes, it is in fact in the IBM JVM on the mainframe.

That means you can run JavaScript on the mainframe. And it will run on the zAAPs. It's pretty trivial to write a Java wrapper that will execute JavaScript code from a batch job. Simple examples are readily available on the internet, and it only takes a little more

[1] http://www.h2database.com/html/main.html
[2] http://helma.org/

effort to figure out how to execute the JavaScript code out of a PDS instead of a ZFS file. Of course REXX is much more closely tied to z/OS, and there are some things that simply can't be done in JavaScript or Java. However, JavaScript definitely has its advantages over REXX as a programming language. I did learn that you need to beware one critical failing though: Rhino's implementation of JavaScript's regular expressions is quite buggy. If you're going to use regular expressions in your JavaScript code under the Rhino included in Java 6, you shouldn't assume that the language interpreter is working properly.

JZOS has been delivered with z/OS for a while now and not only greatly simplifies running Java applications as either batch jobs or started tasks, but also provides a number of useful classes for interacting with z/OS. Creating a Java started task that responds to the expected start and stop operator commands is easy with JZOS. It's only slightly more difficult to add support for modify commands. Need to access traditional z/OS datasets from Java? The delivered JZOS classes make that relatively easy too—including VSAM access, although I haven't tried that yet.

Having said all that, Java is still Java. Personally, I don't particularly care for the language itself, but I like the possibility of running more stuff on the zAAPs, so support for JavaScript within the JVM is very intriguing. And while you might think that interpreting JavaScript within Java would be prohibitively expensive, the overhead seems to be reasonably small and the overhead percentage decreases with increasing amounts of work.

Memory can also be an issue: Java programmers seem to like specifying relatively large JVM heap sizes. That's a more complicated topic, but the short answer is that you might not need as much as you think, unless of course you do.[3] It's definitely an area of concern that you should keep an eye on though: how many 1GB JVMs would it take before you start having paging problems?

But overall, I've been impressed of late with the mainframe's Java support. It runs fast. It runs on the zAAPs. It runs all sorts of Java things without any recoding effort. It can even run other JVM-based languages. (In principle, I think Groovy[4] should run fine, but I haven't tried it.) And you can easily run your Java code in batch, as a started task, from the shell prompt, within WebSphere, or in CICS. So if you haven't tried it recently, you might want to take a fresh look at running Java applications on your mainframe.

So that's what I think I learned this month. If you think I didn't learn my lesson correctly, please email me at sachapman@aep.com and let me know!

---

[3] See http://www.cmg.org/measureit/issues/mit76/m_76_7.pdf wherein I find it's hard to predict heap space.
[4] http://groovy.codehaus.org/