JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 8, September-October 2004

Project Initium: Programmatic Deployment

Douglas Lyon, Fairfield University, Fairfield CT, U.S.A.

In theory,
there is no difference between
theory and practice;
In practice, there is.

-Chuck Reid

Abstract

This paper describes the design and use of a Java Web Start framework called *Initium*. Initium generates a jar file that minimizes the number of included classes by performing a static class dependency analysis. It then prompts the programmer for security parameters that enable the programmatic signing of the jar file, for the purpose of authentication. Initium generates a local Java Network Launch Protocol file (JNLP file) for the purpose of testing, as well as a remote JNLP file, for deployment. Finally, Initium programmatically uploads both the JNLP and jar files to the web server, to complete the deployment cycle.

The signing of a jar file enables web start clients to execute a Java application in a trusted and distributed manner. Trusted jar files can be executed outside of the "sandbox" and thus be given access to files, or be able to open connections to hosts other than the web host on the target system.

Initium is a Latin word that means: "at the start". It is part of an on-going project at both the DocJava Inc. skunk works and Fairfield University.

1 THE DEPLOYMENT PROBLEM

We are given a client-side application, written in Java, which is ready for deployment. Our goal is to find a way to deploy the application to any target desktop, upon demand. The solution to this problem is subject to the constraint that: the application is allowed to run while the target system is offline, the application has no security restrictions, and the application be automatically updated. Further, we seek to minimize the time and effort

"Project Initium: Programmatic Deployment", in *Journal of Object Technology*, vol. 3, no. 8, September-October 2004, pp. 55-69.

needed by the programmer to perform deployment and to minimize the time and effort users need to download the application.

The approach that we use for application deployment is to make use of existing compaction; signing and Java web start technologies. Java web start is a standard adopted by Sun Microsystems that has been bundled with their Java development kit since JDK1.4 [Sun 2004].

The motivation for addressing the deployment of Java is that deployment appears to be the weak link in the development chain. Programmers that attempt to deploy their client-side applications using applets already know the hardship of trying to predict the run-time environment. Distributed computing on an open Internet is fraught with difficulties. How do we up-load our applications to the web server? Should it be a push or a pull technology? How can we make sure that the client has all the classes that it needs in order to run the computation?

There are several possible answers to these questions. To up-load our applications to the web server, I have selected a secure copy mechanism. Other techniques might work as well (or even better). For example, if security is not an issue, ftp might work with less overhead. On the other hand, there are compelling arguments to keep the transmission secure (particularly since a password and user id are sent in the clear, with ftp).

I use a push technology (SCP) to upload my applications, but a pull technology (Java Webstart) to download and run them. In this way, computers that are behind a firewall and are otherwise unreachable, can run the applications. This is the topic of Section 4.2.

In order to make sure that the client has all the classes needed to run the application, I take the approach of creating a jar file. The jar file incorporates all the classes that are *probably* needed. This is done by making use of a “smart” linker, the topic of Section 3.1.

2 TARGET SYSTEM REQUIREMENTS

The first assumption is that the target system has Java web start (JAWS) installed. This is a critical assumption, since without JAWS; deployment using web start technology will fail. In a non-geographically co-located computation environment, the target system may well be located far away and behind a firewall. It is typical for the target system to be running with a non-routable IP address and thus quite difficult to administrate remotely. In fact, such a situation probably describes the majority of target systems in the corporate world. Employees are often prevented from installing current versions of software (Java or otherwise) without a system administrators’ password.

Thus, environment configuration is not just a technical problem; it is a security issue. Further, since Microsoft has elected not to distribute Java with its windows operating system, the issue also becomes political.

Even worse, once JAWS is installed, it must be configured to work. If the configuration is not correct, JAWS will spit out a cryptic error and the user will be left wondering what happened. Clearly, this is not an optimal situation.

We can verify the correctness of environment configuration, before allowing installation to proceed. We can even inform the user of what needs to be installed in order to correctly configure the environment. However, it is beyond our power (or authority) to configure the target system for the user. Installing a current version of Java will generally also install JAWS. However, that may not be sufficient for getting JAWS to work.

2.1 Proxy Webserver Setup

It is typical for companies to filter web downloads and requests through a logging web proxy server. This is designed to monitor web usage. This section describes how to configure a JAWS client that is behind the company proxy web server.

Often, the browser is already pre-configured to use the proxy in order to obtain access to the web. JAWS, however, is not. For target machines behind a proxy web server, you must alter the settings options on the JAWS Management Console. Select "File:Preferences…", as shown in Figure 1.

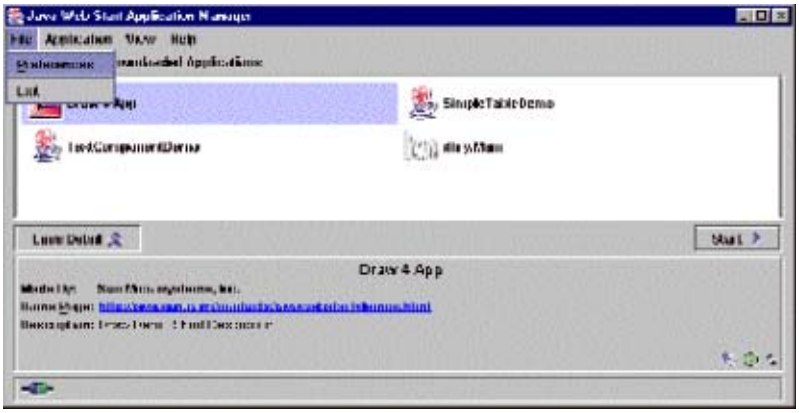


Figure 1. Select *File:Preferences...*

This will cause the *Java Web Start - Preferences* dialog box to appear. Select the *General* tab and select *Use Browser*, as shown in Figure 2.

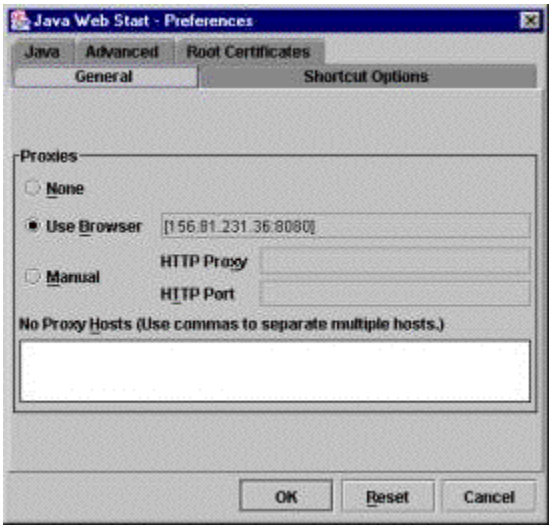


Figure 2. Use the proxy settings in the Browser

Once the setup is completed, the user should not have to alter it again. The question of how to automate this set-up remains open. For example, from within Java, one can tunnel HTTP requests through the proxy web server by putting properties into the *System* class. However, setting these properties is always something that is done at run-time. Generally by some sort of helper method. For example:

```
public static void setHttpProxy(String host, String port) {
   Properties p = System.getProperties();
   p.put("proxySet", "true");
   p.put("proxyHost", host);
   p.put("proxyPort", port);
 }
```

Frequently, a GUI is presented to the user to prompt for these parameters. This could be classified as a network administration task that is being performed by the end-user. Such configuration tasks are cumbersome and error-prone, at best. So far, the best practice is to educate the user of the JAWS technology about proxy web server set-ups, as a *routine part* of deployment! See [Sun 2004a] to learn more about networking properties.

In the simpler case, when users are not required to make use of proxy web servers to gain access to the web, a simpler solution is to make use of a JavaScript that can detect the configuration of the target machine an advise the user accordingly.

2.2 Testing Java Webstart

Testing JAWS is performed by making use of it from a known good JAWS site. For example, demos can be run at http://java.sun.com/products/javawebstart/demos.html.

Some windows users report errors about a bad installation when running JAWS. An example of such an error is shown in Figure 3.

Figure 3. Error during execution

Such users need to uninstall their current version of Java and then re-install it. This can require administrative access. Thus, webstart must be verified and installed on a per-user basis. On the other hand, some systems, like Mac OS X, come pre-installed with JAWS (though testing should still be performed).

3 PROGRAMMATIC DEPLOYMENT

This section describes the 4 steps toward programmatic deployment of a Java application. The first subsection describes how to create a jar file from within a working project so that the jar file only contains those classes that are needed in order to run the application. The second subsection describes how to sign the jar file so that it can be trusted. The third subsection describes how to programmatically synthesize the JNLP script that will launch the application from the local machine, for the purpose of testing. The forth and final subsection describes the interface and testing function available to the programmer.

3.1 Packing the Jar File

Jar deployment is typically performed by programmers who seek to distribute their applications. Typically, programmers will use the Sun-supplied jar tool in order to deploy their applications. It is left, therefore, to the programmer to decide what classes to include and which ones to exclude. For example, my present project, including all the classes, results in a compressed jar file that is nearly 3 MB in size. However, with proper packing and dependency analysis, I can trim this down to 7.2 KB, a improvement by a factor of 40.

How is this possible? How can I take an existing set of 2,106 classes in a project and reduce to just 4 class files (a 500:1 improvement). The answer is in static dependency analysis (SDA). SDA has the advantage of creating very small jar files by being a "smart" linker that does static association discovery. However, SDA *can also fail to work properly*. It is nearly impossible to do static dependency analysis in a language that can dynamically load classes based on the contents of a string. For example:

```
Class c = Class.forName("theClassWeMissed");
```

is missed by the SDA and causes a *ClassNotFoundException* at run time. In my project, *Class.forName* occurs in 34 files out of 1,585 files (1 file in 46). So, while rare, at least in

my own code, I am aware that this does happen. Further, for projects that are smaller than mine, there may be little benefit to packing optimally for size. So SDA is not for every application.

On the other hand, in grid computing, lowering communication overhead can impact how and where we partition a job. It can also impact our use of limited network and CPU resources when a job is distributed and run (that is, we can reduce start-up times). There is a significant computational cost to SDA, but it is paid only once, at upload time, and will speedup every download and all the start-up times, if done correctly.

The *pack task* is an optional task available in *Ant*. Users can invoke the pack task from within Ant, after a proper configuration. However, I am interested in performing my packing programmatically, and as a result, I have created a facade to the pack task. For example:

```
private static void packTask() {
     pack(dhry.Main.class);
     pack(addBk.AddressBook.Main.class);
     pack(pavlik.AffineImagePanel.class);
     pack(classUtils.pack.TestPack.class);
     pack(ip.Main.class);
 }
```

Will create 5 size-optimized jar files containing the *manifest* files that enable the classes to be invoked by the JVM (Java Virtual Machine). It is thus a requirement that the named classes contain a proper *main* method. Invoking the *packTask*, on a 400 Mhz G4, took over 85 seconds for all 5 jar files. There were over 2100 classes to select from, and the output jar files ranged in size from 5kbyte to 750 kbytes. With larger projects, most of the time is spent in dependency analysis. As an added feature, I have created a simplified interface that will input a class and print out all classes that it depends upon. This requires a recursive search, since class *A* can depend on class *B*, and class *B* can depend on class *C*. Thus all classes, *A*, *B* and *C* are formed in the list. For example:

```
public static void main(String[] args) {
     printDependencies(dhry.Main.class);
 }
```

Outputs:

```
dhry.Main
gui.JInfoFrame
dhry.Record
gui.JClosableFrame
```

These are the classes needed to run a Dhrystone benchmark and display the results in a Swing frame. For those programmers that are worried about missing classes, there are options available to direct some classes (and even jar files) to be included. As an alternative, there is always *sun.tools.jar.Main*, which enables the programmer to include all the class files in a given directory, ignoring the SDA.

3.2 Signing the Jar File

This section describes how to programmatically sign a jar file from within a live Java program. This is a far more difficult a problem than it would first appear. There has been some excellent work on the programmatic signing of jar files. Scott Oaks has some code for signing jar files [Oaks 2001]. However, it is not compliant with the JDK tool for signing jar files (called *jarsigner*). In fact Scott confirms this, claiming that programmatic signing of jar files is "problematic" since none of the classes that sign the jar files are public [Oaks 2004].

Raffi Krikorian has an excellent article on signing jar files programmatically, however, it has several problems with the code [Krikorian]. First the code would not compile cleanly, even after applying the bug fix mentioned at http://www.oreillynet.com/cs/user/view/cs_msg/4433. Second, run-time errors appear in the code, preventing actual signing from occurring. The author was contacted, but a bug fix was not forthcoming.

As a hack, and last resort, I made my own mechanism up for signing jar files. Sun has a non-public jar-signing tool in *sun.security.tools.JarSigner*. The only interface to this class is the *main* method. The following code shows a static method for jar signing (based on the *facade* design pattern):

```
public static void sign(String keystoreName,
                        String storepass,
                        String jarFileName,
                        String alias) {
     JarSigner js = new JarSigner();

     String a[] = {
         "-keystore",
         keystoreName,
         "-storepass",
         storepass,
         jarFileName,
         alias

     };
     js.run(a);
 }
```

As of JDK 1.5, the *JarSigner* was removed from the *sun.security.tools* package. This makes a homebrew implementation of the *JarSigner* even more urgent (and so becomes a topic of current research). Scott Oaks [Oaks 2004] has expressed his preference for making use of the *Runtime.exec()* to invoke the *JarSigner* tool.

A jar file can be verified using:

```
public static void verify(String jarFileName) {
     JarSigner js = new JarSigner();
     String a[] = {
         "-verify",
```

```
            "-verbose",
            jarFileName,

        };
        js.run(a);
    }
```

The following code shows how to combine the pack, sign and verification processes into a single invocation:

```
    public static void packSignAndVerify(
            String className,
            String keystoreName,
            String storepass,
            String alias) {
        String jarFileName = className + ".jar";
        TestPack.pack(className,
                      jarFileName);
        sign(keystoreName,
             storepass,
             jarFileName,
             alias);

        verify(jarFileName);
    }
```

One drawback of verification is that it terminates the callers' thread of execution. This is a pity, since verification is just the beginning of what needs to be done at the clients' end (before execution of the jar file begins). Another drawback is that these signing elements are in the Sun private package, and thus can change without notice. A final drawback of this code is that the error output and the information output are hard to parse and directed at the console. Ideally, this should be called through lower-level method invocations, giving the caller a chance to catch exceptions and perform error-correction.

For the purpose of verification, there is a *JarVerifier* in the *java.util.jar* package. However, it is not public. The stable, public API for this is the *java.util.jar.JarFile*, which invokes the *JarVerifier*. Ideally, the *JarVerifier* API should be a public one so that it can be invoked directly. As of JDK1.5, beta, *JarVerifier* is still not public. Probably the safest solution is to make use of the *JarFile* instance and allow the built in verifier to verify the jar file.

3.3 Synthesizing the JNLP Code

This section shows how to reuse the packing and signing parameters to help formulate a Java Network Launch Protocol (JNLP) file. For the purpose of testing, a JNLP file is placed on the local file system with a set of hard-coded parameters. Included in these parameters are the file name, path name, class name, resource requirements, title, vendor, homepage, etc. An example JNLP file, generated programmatically, follows:

```
<jnlp href="dhry.Main.jnlp"
        codebase="file:///Users/lyon/current/java/j4p">
```

```
    <information>
      <title>dhry.Main</title>
      <vendor>DocJava, Inc.</vendor>
      <homepage href="http://www.docjava.com"/>

      <offline-allowed />
    </information>

    <security>
      <all-permissions />
    </security>

    <resources>
      <j2se version="1.4+" />
      <jar href="dhry.Main.jar" />
    </resources>

    <application-desc main-class="dhry.Main" />
  </jnlp>
```

It is both tedious and error-prone for the programmer to have to write these JNLP files. It is much easier, for the programmer, to invoke a simplified interface that synthesizes the JNLP file automatically:

```
public static void writeJnlp(String title,
                             String vendor,
                             String homePage,
                             String jarFileName,
                             String mainClass,
                             File jnlpFile,
                             String codeBase)
```

Many of the *writeJnlp* parameters are reused from the packing and signing process. Thus we are able to pack, sign and synthesize the JNLP file using a single method invocation:

```
    public static void packSignOutputJnlp(
            final String mainClassName,
            final File keyStoreFile,
            final String password,
            final String alias,
            String vendor,
            String url)
```

Programmers can hard-code as many (or as few) of these parameters as is appropriate. For other parameters (like the password), it is probably a good idea to use a GUI to prompt the programmer. ANT enables the hard coding of passwords in the build script (and this is probably a wrong-headed thing to do). As I have a strong preference for keeping passwords out of source code and computer files, I prefer a GUI prompt me for a password, when needed (something ANT does not do).

The following example mixes hard-coded parameters with GUI prompts to keep the program secure, while reducing unneeded interactions:

```
public static void packSignOutputJNLP() {
     final String mainClassName = futils.In.getString(
             "enter class name");
     checkClass(mainClassName);
     final File keyStoreFile = futils.Futil.getReadFile(
             "select keystore");
     String vendor = "DocJava, Inc.";
     String url = "http://www.docjava.com";
     final String alias = "docjava";
     final String password = futils.In.getString(
             "enter password");


     packSignOutputJnlp(mainClassName,
                keyStoreFile,
                password,
                alias,
                vendor,
                url);

 }
```

A simple dialog box is used to prompt the programmer for a password, as shown in Figure 4.

Figure 4. Prompt for a Password

A similar dialog box is used to prompt the programmer for a secure copy password, as described in Section 4.

3.4 Testing the output

Since there is a danger of SDA failure, I strongly suggest that the output of the process be tested before being deployed. The output of Section 3.3 consists of two files, a signed jar file called: *dhry.Main.jar* and a JNLP file called *dhry.Main.jnlp.* JAWS is started when the programmer selects *dhry.Main.jnlp*. A window appears, as shown in Figure 5.

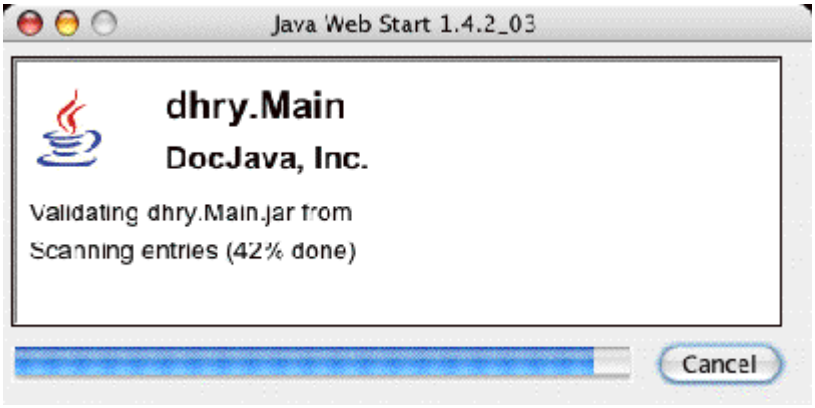


Figure 5. The Jar file Loads on demand

JAWS forms a jar cache and will only reload the jar file if it has been touched since the application was last run. Thus JAWS downloads updates upon demand. After the second run, JAWS will offer to create a shortcut on the desktop.

4 DEPLOYMENT

This section is divided up into two subsections. Section 4.1 describes how to set up a web server for proper file distribution by adding new MIME types to configuration files. Section 4.2 describes how to use the Secure Copy (SCP) API to programmatically upload the files to the web server. While not everyone will have a web server that makes use of secure copy for transferring files, it is probably a good idea to make use of some type of security, and SCP appears to be commonplace.

4.1 Setting up the system for distribution

MIME (Multi-purpose Internet Mail Extensions) is used to permit the transport of non-text message bodies via e-mail. It has been extended to map to a number of different applications. The browser must understand the supplied MIME-TYPE. Often browsers have helper applications or plug-ins that assist them in interpreting data of a specific MIME Type. See http://trade.chonbuk.ac.kr/~leesl/rfc/rfc1521.html for more information about MIME types. JAWS is no different in that a MIME type is associated with JNLP files. These MIME-types help to invoke JAWS, when appropriate.

As pointed out by [Rohaly] and [Sun 2000], running Apache under Fedora (a freely available version of RedHat Linux), requires modification of the mime type list. The */etc/mime.types* file can be altered, but only if you have super-user abilities. The following two lines were added to this file:

```
application/jnlp                jnlp
application/x-java-jnlp-file    jnlp
```

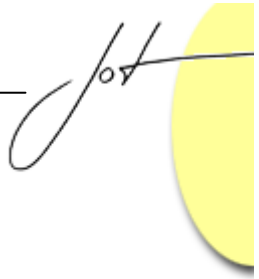
Many modern web servers already incorporate these mime-types in their configuration, and so this step may be unneeded.

4.2 SCP Integration

This section shows how to synthesize a JNLP file suitable for uploading, then shows how to programmatically upload it, using a secure copy, to the web server. It also uploads the jar file needed for deployment. The heart of the secure copy facility is an SSH API available from http://www.jcraft.com/.

An example method that I use for uploading the JNLP files follows:

```
public static void packSignUploadJNLP(String mainClassName) {

        //= futils.In.getString(
        //"enter class name");
        checkClass(mainClassName);
        final File keyStoreFile =
                new File(

        "/Users/lyon/current/java/j4p/keystore");
        //= futils.Futil.getReadFile(
        //"select keystore");
        String vendor = "DocJava, Inc.";
        String url = "http://www.docjava.com";
        final String alias = "docjava";

        final String password = In.getPassword(
                "enter keystore password");


        String codeBase =
        "http://show.docjava.com:8086/book/cgij/code/jnlp/";
        String iconUrl =
        "http://show.docjava.com:8086/consulti/docjava.jpe\"";


        String user = "lyon";
        String host = "192.168.1.95";

        final String rootDirectory =
        "/var/www/html/book/cgij/code/jnlp/";
        String jarFileName = mainClassName +
                             ".jar";

        packSignAndUpload(mainClassName,
                  jarFileName,
                  keyStoreFile,
                  password,
                  alias,
                  vendor,
```



```
                    url,
                    codeBase,
                    iconUrl,
                    user,
                    host,
                    rootDirectory);

  }
```

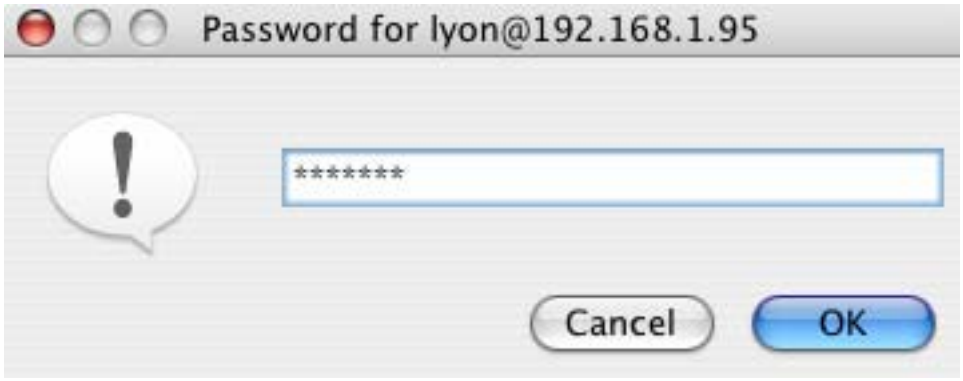


Figure 6. A Simple Password Prompt

Figure 6 shows the second dialog box requiring interaction with the programmer. This password is needed to perform the secure copy, and is also the password needed to access the remote account. As a matter of policy, I decided that no passwords should ever be embedded in the source code (thus requiring some means of input, like a GUI). Further, most of the parameters (like keystore location, home page, etc., are hard coded, where appropriate, to simplify the synthesis of the JNLP, as well as the uploading.

5 CONCLUSION

Programmatic signing is not new [Krikorian]. Nor is the practice of reducing jar size via static dependency analysis [Sadun]. However, integrating deployment (using Java Webstart) along with programmatic signing, jar optimization and secure uploading to a server is new.

One area of possible future work is in the area of programmatic signing of code. It is clear that calling the *sun.util.JarSigner* API is not optimal for several reasons: 1. The class resides in the *sun.util* package, and this package is not generally stable (nor even endorsed for general use!). Additionally, the *sun.util.JarSigner* is only intended to be used from the command line.

The verification mechanism, as implemented in the *JarSigner*, terminates the callers' thread of execution. This is an unexpected, and unwelcome, side effect of a method

invocation. The entire *JarSigner* class needs a rewrite so that it can be invoked from a public API.

In Section 1 we posed several questions that we are now in a position to address: How do we up-load our applications to the compute server? The Initium answer is to enable secure copying over the Internet so that uploads can be accomplished from any location. This requires a specific hard location be known for the destination files, in advance of their synthesis. Perhaps that is not an optimal situation. Hard coded JNLP HREFs are known to change, from time to time, and this can cause fragility. Probably, a better solution, would be to use one of the server-side technologies available to JNLP systems, such as JBoss (http://www.developer.com/java/ent/print.php/3343761).

One of the open problems that remains with JAWS is the set-up problem. Manually setting the proxy web server setting in the JAWS preferences is both error-prone and tedious for users. Worse still, is the long download time needed to install the Java SDK or JRE. Most alarming is the inability to install these things without an administrator's password under Window. To add insult to injury, Windows requires a reboot after the installation (at least under Windows Professional). I have observed that Windows users reboot regularly (I think they actually *like* rebooting).

SDA can fail to work properly. It would be really nice if dynamic dependency analysis could be performed. The question of how to do this remains open. One possible answer might be to log classes as they are being loaded, in order to keep a record of which ones are needed at run-time.

The Initium project has already been used to upload a series of applications. We are working to extend the ideas presented in the paper to help with clusters and grid computing.

The Initium project is an open-source project freely available at http://www.docjava.com.

REFERENCES

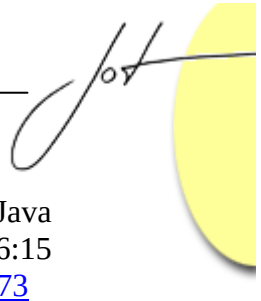
[Krikorian] Raffi Krikorian: "Programmatically Signing Jar Files", *OnJava.com*, http://www.onjava.com/lpt/a/761, April 12, 2001.

[Oaks 2004] Private e-mail correspondence with Scott Oaks, 2004.

[Oaks 2001] Scott Oaks: "Java Security, 2nd Edition", O'Reilly & Associates, Inc., Sebastopol, CA, 2001.

[Rohaly] Tim Rohaly: "Client-side Java makes a comeback", in Javaword, http://www.javaworld.com/javaone00/j1-00-webstart_p.html

[Sadun] Christiano Sadun: "The Ant Pack Task Source Code", open source library available from http://sadun-util.sourceforge.net/



[Sun 2000] Technical Session TS1473 “Introducing Java Web Start: Delivering Java Technology-based Applications Over the Web’ Thursday June 8, 5:15-6:15 p.m.: http://jsp.java.sun.com/javaone/javaone2000/event.jsp?eventId=1473

[Sun 2004] Sun Microsystems: “Java Web Start”, http://java.sun.com/products/javawebstart/developers.html

[Sun 2004a] Sun Microsystems: “Networking Properties”, http://java.sun.com/j2se/1.4.2/docs/guide/net/properties.html

About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. E-mail Dr. Lyon at Lyon@DocJava.com. His website is http://www.DocJava.com.