University of Victoria
Faculty of Engineering
Fall 2006 Work Term Report

Deploying a J2EE Application Using Java WebStart

Department of Physics and Astronomy
University of Victoria
Victoria, British Columbia

Sergey Popov
0121300
Work Term 4
Computer Science/Mathematics
spopov@uvic.ca

December 18, 2006

In partial fulfillment of the requirements of the
Bachelor of Science Degree

Supervisor's Approval: To be completed by Co-op Employer

I approve the release of this report to the University of Victoria for evaluation purposes only.
The report is to be considered (**select one**): ☐ NOT CONFIDENTIAL ☐ CONFIDENTIAL

Signature: _____Position: _____Date:

Name (print):_____E-Mail: _____ Fax #:

If a report is deemed CONFIDENTIAL, a non-disclosure form signed by an evaluator will be faxed to the employer. The report will be destroyed following evaluation. If the report is NOT CONFIDENTIAL, it will be returned to the student following evaluation.

Report Specification

Audience

This report is intended to be read by anyone taking over my position with the Department of Physics and Astronomy, as well as anyone wishing to familiarize themselves with deployment of J2EE applications using Java WebStart.

Prerequisites

Familiarity with Java Virtual Machine (JVM), J2EE applications, XML and web services is assumed on the part of the reader. Some sample Java code and an XML WebStart descriptor file is included in this report.

Purpose

After reading this paper, the reader will gain insight into deployment of J2EE applications over the web using Java WebStart. Furthermore, detailed information on the inner workings of the JVM, especially with respect to the system class loader and security manger components, is provided in this report.

Introduction

The following report describes a major project undertaken by me during my 4 month Coop placement with the Department of Physics and Astronomy at UVic, namely the deployment of an existing J2EE application over the web using Java WebStart technology.

The application in question is called the Job Submission Client, or JSC for short, and is used by users to submit jobs to the computational grid setup by UVic and National Research Council (NRC). The application is written entirely in Java with a Swing GUI and consists of a handful of Jar files in a *.tar.gz* archive. My task was to deploy the application over the web using the Grid Monitor website on which I was also working.

What follows is my account of the process I employed to achieve the stated objectives, ranging from Requirement Gathering to the Implementation stages. Additionally, some sample code is provided to better illustrate the final product, although its understanding is not required due to the analysis provided in this report.

Requirements Gathering

The first stage of my project involved gathering requirements for the final product. The basic question that needed to be answered was "What exactly do I need to build?" This question can only be answered by first addressing the required functionality of the final product. My spec was very simple and open-ended, namely deploy an existing J2EE application over the web.

The application in question requires a very specific environment in order to run, including the client machine having both Java and GT4 installed. Globus, or GT4 for short, is the grid middleware used by the UVic grid and is itself a shorthand way to refer to the Globus Toolkit version 4. GT4 is the *de facto* standard for grid middleware and is used to handle communication, security, data and user management between the different parts of the grid. Currently, Globus supports only the Linux platform. While it is technically possible to use Java WebStart or similar technology to deploy all of the supporting applications as well, it was deemed to be too risky for both time and resources available. Thus, I was to continue under the assumption that the supporting applications would already be installed and configured on the client machines.

Lastly, there was a single functional requirement, namely that, given the environment specified above, the only thing the user would need to do to deploy JSC is to click a link on a web page and accept the security certificate. The downloading, installation and linking of the JSC with the supporting application would all be done automatically using Java WebStart. WebStart would also be used to launch the application once configured and then pass the control to JSC itself.

Given such usability and functionality goals, it was up to me to make it happen. I started

by looking at the available technologies and seeing if I could somehow piece them together to achieve the desired result.

Technology Exploration

After finalizing what it is that needed to be built, my next task was to find a way to build it. This section will talk about the different concepts and technologies that I considered and eventually used to achieve said goal.

The J2EE Application

First, I had to make sure that it would be feasible to deploy the target J2EE application using Java WebStart. Luckily for me, the Job Submission Client was an in-house create app in the later stages of the development cycle and, most importantly, I had access to the lead developer. Thus it was fairly easy to get all of the required environment variables and command line options that I would need in order to launch the application properly.

The biggest risk item was the application's need to have write access to the user's filesystem in order to persist job information. However, I could not find any information about whether or not this was possible, other than a cryptic note about the security attribute in the WebStart descriptor file.

Webstart Descriptor File

Java WebStart is bundled with each distribution of the JRE and is automatically installed on the client's machine with the JRE. However, before WebStart can launch a remote application, it needs to know some things about the program. This is where the WebStart descriptor file is used.

The descriptor file specifies the location of the application Jars on the remote machine, the minimal version of the JRE that has to be installed on the client machine in order to run the application, as well as the entry point or main class and any security requirements of the app. This file is the only piece of information needed by WebStart to automatically download the Jars, configure the environment and launch the application in the default console.

The WebStart descriptor file that I constructed is provided in Appendix I as *jsc.jnlp*.

Apache Support

In order to deploy using WebStart, there is an additional configuration step that needs to be done to the Apache server such that Apache will associate WebStart descriptor files with Java WebStart. It involves adding one line to the Apache configuration file *mime.types* as the following:

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

Self-Signed Jars

WebStart provides the ability to run potentially any code via the web and, in fact, will go so far as to launch the application in the default JVM. Consequently, some security mechanism needs to be in place to prevent damage to unscrupulous users' machines by having them simply click a Java WebStart link. The current implementation of WebStart requires that all jar files specified in the WebStart Descriptor file be signed with the same security certificate that the client has to accept as valid before the program can run.

Ideally, these certificates would come from a trusted and central authority like VeriSign or Grid Canada; however, for development-only purposes I decided to use a self-signed

certificate that I created using Java's packaged *jarsigner* application. Since the use of the *jarsigner* utility is outside the scope of this essay, the reader is encouraged to explore this topic on their own.

Prototyping with HelloWorld

Due to the large number of new and poorly understood technologies involved, I decided to construct a prototype WebStart application to test my understanding of the various components and their interactions. As a starting point, I went to the Java tutorial section of Sun's website and copy-pasted the Java Swing *Hello World* tutorial code. I then compiled the code, created a *HelloWorld.jar* Jar file, signed it using the above procedure, created a new WebStart descriptor file that would call the main method in the new Jar and copied both of them over to the development Apache environment.

To verify that everything works as I expected, I simply pointed my browser to the *.jnlp* file, clicked *[Accept]* on the security warning prompt and watched *Hello World* in all its glory. The success of the prototype suggested that it is indeed possible to deploy a J2EE GUI application over the web. One of the fallouts from the prototyping stage is my discovery that WebStart uses its own classpath and JVM options that are different from the corresponding client machine variables. Thus, my next step was to create a wrapper that would initialize the Java environment for the Job Submission Client.

WebStart Wrapper

After the requirements gathering and prototyping stages, it was now time to start the actual implementation. In this case, the *WebstartWrapper* class that will be called by Java WebStart to setup the environment and load the Globus jars before handing the control over to the JSC. This was an important and required step since deploying with WebStart is not the same thing as launching the app from a shell script due to WebStart's use of its own Java classpath with its own JVM options.

Java Environment Variables

From the Requirements Gathering phase, I knew that there were two environment variables needed by the JSC and not included in the JVM as launched by WebStart. These were *GLOBUS_LOCATION*, a shell variable that points to the root installation directory of GT4, and *axis.ClientConfigFile*, a web services definition file stored with a specific offset from *GLOBUS_LOCATION*.

Initially, I tried using the *System.getenv()* call to read the *GLOBUS_LOCATION* shell variable, but it was inconsistently throwing security exceptions. To get the shell variable, I created a new subprocess that ran the *env* shell command and parsed the output. The web services definition file offset was determined to be constant for all installations of GT4 and can be simply concatenated to the *GLOBUS_LOCATION*.

Finally, I used the *System.setProperty()* call to add these two variables to the JVM.

Java Classpath

The requirements gathering phase revealed that the JSC needs to have access to the GT4 libraries in order to function properly. However, since Globus can be installed in an

arbitrary location, it is not possible to use a WebStart descriptor file to specify a static location like one might do with a shell launch file. The classpath needed to be changed at runtime.

Unfortunately, Java 1.5 does not provide a natural way to dynamically change the library path programatically. Instead, the assumption is that you will use the *-cp* command line switch when starting the JVM. Hence, my initial attempt at solving this problem consisted of creating a subprocess – in much the same way as reading shell variables – and launching a new JVM instance using the command line arguments to set the classpath to look in the GT4 install directory.

This did not turn out to be a good solution because of the way subprocesses are created in Java. Java subprocesses created with *Runtime.exec()* do not share the parent process' console or terminal session. In fact, the child process creates a new instance of both *InputStream* and *OutputStream* objects that are completely separate from *System.out* and *System.in* streams. Effectively, this means that a substantial amount of stream redirect code would need to be written in order to satisfy the user interaction component of the JSC UI.

Luckily for me, a Google search revealed the following page: http://forum.java.sun.com/thread.jspa?threadID=300557. In this posting, a very clever person has figured out the way to access and modify the system class loader to load a single external .jar file using the Java Reflections API. The big advantage of using the system class loader (over creating a custom and separate instance of the generic *ClassLoader*) is that libraries loaded with the system loader can be accessed as if they were included in the classpath command line argument. In other words, the code provided is able to dynamically load a single Java jar file as if it has been specified in the original command

line argument to start the JVM without the need to constantly reference *ClassLoader* objects for each library call. This approach provided a big advantage in that the JSC code could be left as is and a separate branch would not need to be maintained for WebStart deployments.

One last problem remained, namely that GT4 contains no fewer than 101 separate jar files and most, if not all, of them are used by JSC. Building on the above code provided in the Java forums on Sun's website, I was able to make a trivial change in scope to create a *ClassPathUpdater* class that could load an arbitrary number of individual Jar files using the system class loader at runtime.

Java SecurityContextManager Class

The last hoop that needed to be jumped through was with respect to the JVM permissions management. Java WebStart launched JVMs use a default *SecurityContextManager* class to verify that the application has appropriate permissions to access certain system resources. This check is in addition to security certificates and signed jar files, and is implemented by artificially issuing a *SecurityManager.checkPermission()* call with the corresponding system resource and action request before each system call. If the specified access to the named resource was allowed, the method would simply return. Otherwise it would throw a security exception and halt execution.

As shown earlier, in the Java Environment Variables section, for some unknown reason this *checkPermission()* call disallows access to system properties, including those properties needed by JSC and explicitly set in the earlier steps.

Since time pressure was beginning to be more and more of a factor in decision making, I

decided to create a dummy class that would bypass the security check by simply returning on any input to the *checkPermission()* method. I then used the *System.setSecurityManager ()* call to overwrite the default security manager.

Threat Analysis

A rough and ready threat analysis failed to reveal an attack vector to the potential security hole opened by bypassing the system security manager since the executing code still runs within the JVM memory sandbox, thus mitigating against any buffer overflow attacks. Furthermore, local permission elevation attacks will still need to circumvent the Globus API requirement for a valid grid proxy certificate in order to penetrate the JSC on any ports it may have opened.

Packaging

Once the wrapper was built and tested, a packaging decision needed to be made with respect to the location in the repository that will house the new code. The issue at hand is that while there were separate locations for the JSC and web portal code bases, the *WebstartWrapper* code more of less spanned the gap between them. Logically, the wrapper was part of the JSC application in the sense that it added new functionality to the application by allowing it to be web-deployable. On the other hand, the wrapper code was extra to the core JSC code and specification and acted as a "glue" to bind the JSC and the web portal.

In the end, the deployment decisions dictated that the wrapper code be part of the Grid Monitor project and was checked into the portal repository.

Deployment

Deployment decisions centered around the need for maintaining a "one click build and deploy" environment in the Grid Monitor project. Since the scope of said policy is tangent to the scope of this essay, I will only go so far as to state that Ant and Eclipse were used to enable this functionality while the functionality itself proved to shorten both the development and the release cycles.

Thus, the project build script needed to be modified to allow for the simultaneous build and deployment of the *WebstartWrapper* code with the rest of the Grid Monitor web portal. The build part required not only the compilation of the various class files, but also the creation and security certificate signing of the *WebstartWrapper* jar file. While the deployment section needed only a single modification to include the new webstart directory that housed the WebStart descriptor file and all of the signed jars needed by WebStart. An additional anchor tag was added to the portal HTML to link the WebStart descriptor file for online deployment of the Job Submission Client.

Because the wrapper code was tested and verified as working before it was added to the project build, no issues were experienced due to the above changes. Those developers that did not need to work with the new code were able to easily exclude the WebStart target from their build scripts.

Summary/Conclusion

In conclusion, deploying a J2EE application over the web is very possible, but not trivial. In my experience, the complexity of the target application has a lot to do with the complexity of the deployment. In the case of the Job Submission Client, a number of changes to the Java Virtual Machine had to be made at runtime in order to properly bootstrap the JSC to the client's environment.

First, all of the application jars needed to be signed with a security certificate. Second, the Java classpath needed to be changed at runtime in order to load needed system libraries whose location was unknown until the program actually ran. Lastly, Java security model needed to be altered to allow the application access to the client's filesystem such that the application could overcome unexpected termination and potential data loss by persisting its state to disk.

After completing this project, I have acquired additional knowledge of the inner workings of the Java Virtual Machine. Specifically with respect to the class loader processes and the system security model. I have also gained first hand experience with deployment of J2EE applications over the web using Java WebStart.

Appendix I

This appendix contains some sample code and configuration files that were used in this project.

jsc.jnlp

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JSC Web Start Deployment Template -->
<jnlp spec="1.0+"
   codebase="http://ugdev03.phys.uvic.ca/SergeySandbox/webstart"
   href="jsc.jnlp">

<information>
   <title>GridX1 Job Submission Client</title>
   <vendor>phys.uvic.ca</vendor>
   <homepage href="http://www.grid.phys.uvic.ca/index.html"/>
   <description>Job Submission Client</description>
   <description kind="short">
      Submit grid jobs via the web.
   </description>
   <icon width="160" height="96" href="../images/grid-bg-color.jpg"/>
   <icon kind="splash" href="../images/banner_left.gif"/>
   <offline-allowed/>
</information>

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

<resources>
   <j2se href="http://java.sun.com/products/autodl/j2se" onclick="javascript:mytracker(this.href);"
version="1.5+" />
   <jar href="webstart.jar"/>
   <jar href="JobSubmissionClient.jar"/>
   <jar href="commons-io-1.2.jar"/>
   <jar href="commons-lang-2.2.jar"/>
   <jar href="jhall.jar"/>
   <jar href="derby.jar"/>
   <jar href="derbytools.jar"/>
   <jar href="jlfgr-1_0.jar"/>
   <jar href="jsamlib.jar"/>
</resource>

<application-desc main-class="ca.gridx2.webstart.WebstartWrapper">
</application-desc>

</jnlp>
```

WebstartWrapper.java

```
/**
 * WebstartWapper.java
 * Created Nov 27, 2006
 *
 * This is a wrapper class used to launch the Job Submission client using
 * Java webstart.  It reads in all of the required shell variables and passes
 * them off to the JVM before launching the main method of the JobSubmissionClient
 * class, because webstart does not appear to propogate shell variables.
 **/

package ca.gridx2.webstart;
import java.io.*;
import java.util.*;
import java.net.*;

public class WebstartWrapper {

	public static void main(String[] args) throws IOException{
		//get Globus information
		String globusHome = getEnv("GLOBUS_HOME");
		if(globusHome.equals(""))
		   globusHome = getEnv("GLOBUS_LOCATION");

		//munge the classpath
		loadGlobusJars(globusHome +"/lib");

		//set JSC specific properties
		System.setProperty("GLOBUS_LOCATION", globusHome);
		System.setProperty("axis.ClientConfigFile", globusHome +"/client-config.wsdd");

		//launch the main app
		String[] toss = new String[1];
		toss[0] = "-gui";
		try{
			System.setSecurityManager(new WebstartSecurityManager());
			ca.gridx1.sp.JobSubmissionClient.JobSubmissionClient.main(toss);
		}catch(Throwable t){
			System.err.println("Unable to run app because of: " +t.toString());
			t.printStackTrace(System.err);
			throw new IOException("Fatal Error! " +t.toString());
		}
	}//end method

	// workaround for security violations being thrown when accessing
	// System.getenv()
	private static String getEnv(String name) throws IOException{
		Process p = null;

		try{
			p = Runtime.getRuntime().exec("env");
		}catch (IOException e){
			throw new IOException("getEnv returned with an error: " +e.toString());
		}

		BufferedReader br = new BufferedReader(
			new InputStreamReader( p.getInputStream() )
		);

		String toss = br.readLine();
		while( toss != null ){
			int idx = toss.indexOf('=');
			String key = toss.substring(0, idx);
			//System.out.println("Comparing: " +key +" to " +name);
			if( key.equals(name) )
				return toss.substring(idx+1);

			toss = br.readLine();
		}//end while

		return "";
	}//end method

	private static void loadGlobusJars(String libPath) throws IOException{
		File dir = new File(libPath);
		File[] allFiles = dir.listFiles();

		for(int i=0; i<allFiles.length; i++){
			if( allFiles[i].getName().endsWith(".jar"))
				ClassPathUpdater.addFile(allFiles[i]);
		}//end for

	}//end method
}
```

ClassPathUpdater.java

```
//most of the code borrowed from:
//"http://forum.java.sun.com/thread.jspa?threadID=300557"

package ca.gridx2.webstart;
import java.io.*;
import java.net.*;
import java.lang.reflect.*;

public class ClassPathUpdater {
        public static void addFile(String s) throws IOException {
                File f = new File(s);
                addFile(f);
        }//end method

        public static void addFile(File f) throws IOException {
                addURL(f.toURL());
        }//end method


        public static void addURL(URL u) throws IOException {
           //local scope so that we can load multiple files at once
           final Class[] parameters = new Class[]{URL.class};

           URLClassLoader sysloader = (URLClassLoader)ClassLoader.getSystemClassLoader();
           Class sysclass = URLClassLoader.class;

           try {
                Method method = sysclass.getDeclaredMethod("addURL",parameters);
                method.setAccessible(true);
                method.invoke(sysloader,new Object[]{ u });
           } catch (Throwable t) {
                t.printStackTrace();
                throw new IOException("Error, could not add URL to system classloader");
           }//end try catch
        }//end method

}//end class
```