A Heterogeneous Screen-Saver for CPU Scavenging

Douglas A. Lyon, *Senior Member, IEEE,* Pawel Krepsztul, and Francisco Castellanos

Abstract— **This paper describes an on-going project known as the *Initium RJS* (Remote Job Submission) system. The system mixes Java and native technologies to create a CPU-scavenging, hetrogeneous, screen-saver-based grid computing system.**

Web start technologies, along with a unique upload and deployment technique, enable the semi-automatic installation of screen-savers. The *Initium Wizard* makes use of static dependency analysis to generate a jar file that minimizes the number of included classes. It prompts the programmer for security parameters that enable the automatic signing of the jar file for the purpose of authentication. Initium generates a Java Network Launch Protocol file (JNLP file) and automatically uploads both the JNLP and jar files to the web server.

The signing of a jar file enables screen-saver initiated web-start clients to execute a Java application in a trusted and distributed manner. Trusted jar files execute outside of the "sandbox". This supplies an autonomic feature that enables CPU scavenging by the grid.

A screen-saver technology controls a Webstart application. The Webstart application uses a multi-cast query to locate a look up server (LUS). The LUS uses RMI/SSL to contact a Web Sever (WS) that collects grid information and dispatches jobs. The LUS sends results to the WS for later harvesting by the grid user.

We are interested in Screen-savers because they represent a minimally-invasive technology for volunteering CPU services. A computer that lacks a screen saver has a utilization that is typically between 40 and 60 hours out of a 168-hour week (i.e., 35% of the time). As Java programmers, we have found no work in the area of Java-based screen-savers with an eye toward grid computing, and thus we feel our efforts in this area are novel.

Java provides a heterogeneous compute environment and, by extension, a screen-saver framework ported to a variety of platforms should enable a heterogeneous volunteer army of CPUs upon which a grid may scavenge. We address several sub-problems related to this effort, including deployment, security and discovery.

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.

Manuscript received June 4, 2006.

Dr. Lyon is Chairman of the Computer Engineering Department at Fairfield University, Fairfield, CT 06824 USA phone: 203-641-6293; fax: 203-877-4187; e-mail: lyon@docjava.com.

Pawel Krepsztul is with Pepsi Bottling Group, Inc., 1 Pepsi Way Somers, NY 10589 USA; e-mail: Pawel.Krepsztul@pepsi.com).

Francisco Castellanos is with the Computer Engineering Department at Fairfield University, Fairfield, CT 06824 USA.

Index Terms— **Distributed Computing, GridComputing, Java, Object-Oriented Programming, RMI, Screen Savers**

I. Introduction

This paper describes the middleware needed to deploy jobs to non-geographically co-located clusters with decentralized look-up severs. The system includes a hybrid screen-saver system that installs on multiple platforms (Mac, Windows and Linux). The Screen-Saver (SS) helps with the CPU scavenging. We have named our framework the Initium Remote Job Submission (RJS) system. Initium generates a minimal-sized, signed jar file [Lyon 2005c][1]. A Computation Server (CS), (a remote computer running the Initium Compute Server Software), runs the jar. A Web Server (WS) has a Java Network Launch Protocol (JNLP) file that references the jar file on the WS. The signed jar file contains a job for the CS, known as the *computation jar.* A Look Up Server (LUS) is an application that runs on a server within a LAN and provides resource management. The LUS pushes the JNLP link to an available CS. The CS executes the JWS job and transmits the answer back to the LUS using RMI over SSL (RMI/SSL). The CS registers with the LUS using multi-cast IP packets. The Web Start-initiated LUS then updates its list of computation servers.

The *Initium Wizard* (IW) deploys the screen-savers to the WS so that people who wish to volunteer their computers into the grid may do so with reduced effort. The grid programmers who wish to upload their Java application into the Initium RJS system also use the IW.

The motivation for addressing the deployment of Java is that deployment appears to be the weak link in the grid computing development chain. From the user point-of-view, a computer needs to be available when the user needs it. Our goal is to provide a minimally invasive CPU scavenging technology. The IRJS screensaver activates during user-computer quiescence. The converse is also true, when the period of user-computer quiescence ceases, the screensaver terminates any currently running compute jobs, releasing the computer back for general use. Such a program constitutes a first step toward utilizing otherwise idle compute resources in a grid computing system.

Initium uses a push technology (SCP) to deploy applications to the WS. The SS uses Java Web Start (JAWS) to download and run them. In this way, computers that are behind a firewall and are otherwise unreachable can use the Initium system.

In order to make sure that the client has all the classes

needed to run the application, Initium does static dependency analysis on the Java byte codes.

The following sections describe the target system requirements, the jar packing process, resource management, security, screen savers and the RJS middleware sub-system.

II. Target System Requirements

Java Webstart installation is a prerequisite for participation in the Initium RJS grid. We also require that the grid programmer have a password to the Initium RJS WS and the ability to upload, using SCP (i.e., transmit port 22 traffic to the WS).

Normally, a user can install a screen-saver. Screen-savers (under MS Windows) go in the *Windows32 system* directory. Sometimes system administrators deny access to this directory and this is a limiting factor. Linux and MacOsX do not have this limitation.

Another limitation is in network communications. Companies sometimes filter web downloads and requests through a logging web-proxy server, designed to monitor web usage. The browser automatically obtains the proxy configuration for web access. JAWS does not accomplish this task easily. For target machines behind a proxy web server, the user must alter the options on the JAWS Management Console. The question of how to automate this set-up remains open. The user must often enter these parameters by hand (i.e., the user is performing a cumbersome and error-prone network-administration task). So far, best practice is to educate the user about proxy web-server set-ups, as a *routine part* of deployment! To learn more about networking properties see [Sun 2004a][2].

The afore-mentioned problem represents an impediment to deployment and remain open. We theorize that the JXTA framework might help solve the problem, but have not investigated it, yet.

III. Packing the Jar File

One sub-problem in remote job submission is to trim the job down into a small, self-contained, signed jar file. The Initium RJS system makes used of static dependency analysis (SDA) in order to reduce the size of the file. We show that this is a deep problem in grid computing, as improperly configured resources cause a job to fail after deployment into a grid. Such failure can be hard to debug and correct.

The question of how effective SDA can be remains open. We have conducted a study of 130 deployed web start applications, available for evaluation at http://www.docjava.com.

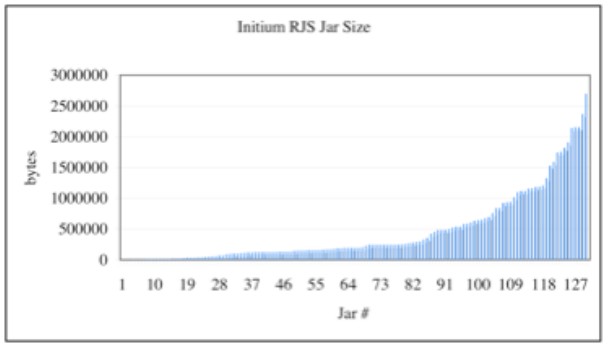


Fig. 1. Jar size vs. Jar Number.

Fig. 1 shows the size of the jar file, in bytes. The largest jar is 2.7 MB, the smallest 2.3 KB and the average is 458 KB. Thus, on average, we have been able to achieve a compression ratio of almost 6:1 (with a maximum of over 1000:1).

Compression ratios for jar size reductions are hard to predict. More onerous is that SDA can *fail to work properly*. It is hard 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 our project, *Class.forName* occurs in 34 files out of 1,585 files (1 file in 46). Further, for smaller projects, there may be little benefit to packing optimally for size. Thus, SDA is not for every application.

The SDA is now a "smart" linker, but it has limits. We use our domain knowledge about different applications to decide what resources should and should not be included. For example, most sane implementations of the Java virtual machine have API's available that include all classes in *java.lang*. However, some implementations (e.g. micro editions that run on PDA's and cell phones) lack *javax.swing*. Thus, we direct the SDA not to include the swing classes as a part of a standard deployment, by default. However, if our assumptions regarding the run-time environment prove erroneous, our deployment will fail with a *classNotFoundException* thrown at run-time.

Other, less standard resources are collected and stored in a series of support jar files that reside on the web server. This too, poses significant deployment issues, which we shall address, below.

In the case of reflection (which is able to find classes from a string), SDA is not technically sufficient. To address this concern, we theorize that dynamic dependency analysis (DDA), or, at the very least, a scan of all SDA invocations to *Class.forName(String s)* would be needed to help the SDA. At least that would give us the needed class files. The problem is, the system only loads class files, not resources (like icons, etc.).

If you use the *part* and *package* mechanisms of Java web start to declare what packages are included and in what jar

files they reside, you should be able to skip downloading of any jars declared as *lazy*. This should work, *in theory*. However, in practice, it does not work (as of JDK1.5, or our latest attempt with JDK1.6), and without jar Indexing implemented, there is no way to prevent all jars from downloading. The reason why is that the class loader cannot know where to look for a resource when confronted with a list of jars.

Under JDK 6, *part* and *package* mechanisms are supposed to be working (and perhaps this is true, just not for us!). The theory is that the new indexing option of the *jartool* enables the construction of a master index. This index contains a list of all resources in all jars. The class loader loads this first and uses it to locate resources at run-time. Summary: if the first eager jar contains a proper jar Index of the complete set of jars, then if all of the other jars are marked lazy, they will only be downloaded when the *JNLPClassLoader* requests a resource or class in them. To understand the importance of the situation, consider the following JNLP test file:

```
<jnlp
   href="bookExamples.ch24Reflection.Load
   TestLarge.jnlp"
codebase="http://show.docjava.com:8086/b
   ook/cgij/code/jnlp/">
  <information>

   <title>bookExamples.ch24Reflection.Loa
   dTestLarge</title>
<vendor>DocJava, Inc.</vendor>
     <homepage
   href="http://www.docjava.com"/>
         <icon
   href="http://show.docjava.com:8086/con
   sulti/docjava.jpe"/>
    <offline-allowed />
  </information>
<security>
  <all-permissions />
</security>
<resources>
<j2se version="1.5+" />
<jar
   href="bookExamples.ch24Reflection.Load
   Test.jar" />
(40 jar hrefs later)…
</resources>
<application-desc main-
   class="bookExamples.ch24Reflection.Loa
   dTest" />
 </jnlp>
```

The *lazy* directive tells the webstart client that it should only load the 22 MBs worth of jars if it needs them (add another 10 MBs worth of jars if you want voice synthesis). The above loads eagerly, and a slow down-load kills fast start-up. The *LoadTest.jar* is less than 3K bytes. The *lazy*-tag bug work-around is to strip out the unneeded jar files, by hand, and create a leaner JNLP file (this error-prone and tedious, *get-er-done* activity appears to be industry-standard practice!). For example:

```
<jnlp
   href="bookExamples.ch24Reflection.Load
   Test.jnlp"
codebase="http://show.docjava.com:8086/b
   ook/cgij/code/jnlp/">
<information>
<title>bookExamples.ch24Reflection.LoadT
   est</title>
    <vendor>DocJava, Inc.</vendor>
    <homepage
   href="http://www.docjava.com"/>
<icon
   href="http://show.docjava.com:8086/con
   sulti/docjava.jpe"/>
<offline-allowed />
  </information>
<security>
    <all-permissions />
  </security>
<resources>
   <j2se version="1.5+" />
<jar href=
"bookExamples.ch24Reflection.LoadTest.ja
   r" />
  </resources>

  <application-desc main-
   class="bookExamples.ch24Reflection.Loa
   dTest" />
</jnlp>
```

The reader can try this at home using:
http://show.docjava.com:8086/book/cgij/code/jnlp/bookExamples.ch24Reflection.LoadTest.jnlp

Now the pathological advisory devises a counter example. Try the "same" program using:
http://show.docjava.com:8086/book/cgij/code/jnlp/bookExamples.ch24Reflection.LoadTestLarge.jnlp

The hand-edited JNLP file downloads 200 times faster, but suffers from an ad-hoc/manual synthesis technique. Thus, we have sacrificed speed for reliability (which we judge to be a poor trade-off). We assert that correct and automatic packaging is critical to deployment-scheme success. It is not rational to expect grid-program authors to hand-edit JNLP files without error. The resulting exceptions are sure to be cryptic to most programmers.

This problem is not just endemic of Java operating environments. We have often seen students who were unable to demo a program because they forgot a shared library (i.e., a DLL) from their home system.

On the bright side, once downloaded, the jar files they need not be downloaded again (unless they change). On the other hand, all the jars change at least once per year, since certificates used to sign jars expire after one year.

We created a master index of each jar file, along with its contents, using:

```
jar i catalog.jar *.jar
```

This created a file, in the jar file, called *INDEX.LIST*. A new jar file, added to the head of the JNLP, has an eager download:

```
<resources>
    <j2se version="1.5+" />
    <jar
   href="bookExamples.ch24Reflection.Load
   Test.jar" />
    <jar href="libs/catalog.jar"
   download="eager"/>
```

However, we find that this does not speed start-up either. For example, the hand-optimized JNLP file ran in just 3 seconds from the start of a click (using a 100 Mbps LAN

from a 2.7 GHz Linux box). The new catalog instrumented load tester ran after 27 seconds, each jar downloaded *even though they were not required*. An upgrade to the newest beta 2 version of JDK 1.6 did not help matters at all.

As bug remains, our present effort takes the approach of implementing our own catalog synthesizer and catalog reader. The next step in the research agenda is to integrate this into the JNLP synthesis stage. We suspect that comparison between required resources and given jar libraries can help to synthesize correct JNLP files, automatically. However, this line of exploration is not complete and thus, the problem remains open.

IV. Resource Management

Decoupling data from source code can be a fruitful source of fragility. For example, suppose that you write a program that seeks to make use of an icon in an interface. In order to load the icon, you write:

```
LookAndFeel.makeIcon(getClass(),
   "icons/ColorIcon.gif"));
```

Now suppose the GIF "ColorIcon" icon file is relocated, relative to the root of the source code. This can easily happen during the process of distribution or development. Even worse, resource deficiencies cause run-time errors (perhaps days or even weeks after deployment).

While SDA can identify most of the needed classes, it cannot identify the resources needed by the classes. Resources for a modern application are typically data files (i.e., sound, data-bases, images, image sequences, 3D data, etc.).

Initium has a technique for integrating programs and their resources. The goal is to distribute the programs to a variety of platforms without losing the resources that they need in order to run. Programs so integrated are less fragile than their non-integrated counterparts. The technique uses a semi-automatic source code synthesizer, XML-based serialization and a base-64 GZIP encoded string format.

The approach is suitable for small data objects (i.e., icons, short audio signals, native libraries, etc.). One drawback of the technique is that an added step is required during program development in order to integrate resources into the code. Another drawback is that integrating resources into the source code can dramatically increase the size of the class files. On the other hand, once the class files are loaded, the resources are available in memory (and hence, quickly accessible).

By integrating resources with the source code, the compiler makes sure that the resources are present. In this way, we trade-off a run-time error for a compile-time error.

The result is a self-contained resource without the normal source of fragility (i.e., source relocation). Hence, we no longer have a program that requires files to be located in particular places on the disk. This works well for short data files. However, longer data files lead to longer strings and this results in a compilation error (a "constant string too long" error).

A resource manager combats the error by automatically sensing an updated jar file on the web. The jar file contains data (with no executable code) and downloads into a specified location on disk, after prompting the user for permission. The question of how to resolve this issue, in a more automatic way remains open.

The JNLP native tags address the question of how to deal with native libraries. As an example, consider the *JAddressBook* application, capable of dialing the phone via a serial-port based modem. Modern macs have no serial port and thus a USB to serial adapter is used. The present implementation uploads multi-platform based native method serial port drivers, along with their relevant JNLP entries. An excerpt from the JNLP file follows:

```
<resources os="Mac OS X" >
<jar href=
   "libs/rxtx/mac/RXTXcomm.jar"
  download="eager" />
<nativelib href=
      "libs/rxtx/mac/native.jar"
  download="eager"/>
</resources>
<resources os="Linux" >
<jar href="libs/rxtx/linux/RXTXcomm.jar"
  download="lazy" />
   <nativelib
  href="libs/rxtx/linux/native.jar" />
</resources>
<resources os="Windows XP" >
<jar href="libs/windows/RXTXcomm.jar"
  download="eager"/>
<nativelib
  href="libs/windows/native.jar" />
</resources>
```

This defeats the notion of having a single, integrated jar file with all the resources embedded. In order to bring that back into the fold, a means is needed to gain access to the *java.library.path* at run time, so that native methods can be placed in the proper place, with their versions controlled. In order to provide write access to the library path property, we make use of reflection to alter the visibility of the member variable. Hence the Ugly Gaudy Hack (UGH) :

```
public static void ugh(){
 Class loaderClass = ClassLoader.class;
 Field userPaths =
  loaderClass.getDeclaredField("sys_path
  s");
 userPaths.setAccessible(true);
 userPaths.set(null, null);
 userPaths.setAccessible(true);
 userPaths.set(null, null);
}
```

A code inspection of the core Java API shows that altering the *java.library.path* will otherwise make no difference on which native libraries are loaded (without UGH). Now that we have UGH control over the *java.library.path*, we install bundled native method resources into writable locations. Thus, the native method is UUEncoded into a Java string and compiled as a part of the code. For example:

```
static String librxtxSerialDotsoName =
   "librxtxSerial.so";
static String librxtxSerialDotso =
"H4sIAAAAAAAAAO19DXRURdJoJwQJEJ2AqKgoI4K
   CQggR/wAlhAwQTWAMiaKiw5CZZAa
SmXF+" +.....
```

```
public static void writeLibrary() throws
   IOException {
ByteArrayOutputStream baos = new
   ByteArrayOutputStream();
ObjectOutputStream oos = new
   ObjectOutputStream(baos);
oos.writeObject(Base64.decodeToObject(li
   brxtxSerialDotso));
oos.close();
Futil.writeBytes(SafeCommDriver.getRxtxL
   ibFile(), baos.toByteArray());
}
```

The resource encoder is a part of the Initium RJS system, and helps to automate the process of deployment. The drawback is that the code grows by the size of the native libraries that it must now carry. For the serial port libraries, that is about 56 KB per platform.

V. The Thawte Web of Trust

In order for Java Web Start (JAWS) to give unrestricted permission for a Java program to execute, it must use a "signed" jar file. A signed jar file authenticates the code originator. This does NOT prevent the author from writing harmful code. On the other hand, if you trust the author to write non-harmful code, you may feel safer about running the authors' programs.

In order to sign a jar file, you need a digital certificate. A Certification Authority (CA) issues certificates after a proper background check. Running applications with un-trusted signatures (i.e., a signature that is not verified by a known CA) will initiate a dialog: "It is highly recommended not to install and run this code".

Grid operators want assurance that a program is safe. Proper attribution to the program author does not ensure safety, but it does assign responsibility. In the event, the grid application contained damaging code, the compute servers on the grid become infected faster than normally propagated computer contagions. Such a program places the entire grid as risk.

Grid computing services have a choice. One approach is to require that every user of the grid obtain a certificate. When a new certificate appears on the grid, the computer servers (when in screen-saver mode) will then query the CS owner if the application is to be trusted. We find this to be a bit of a showstopper. Another approach is to sign the jar file with our own certificate. As an alternative, the *Jparss* approach is to issue a temporary certificate to a user that has a certificate issued by a valid CA [Chen][3].

Still another approach is the signing of applications that are unsigned, in the WS. The basic idea is that the grid operator knows the user, since they already have an account. We have yet to find another grid system that has taken this approach. Perhaps the most obvious reason is that it places the credibility of the signer on the line. If you sign off on a Trojan horse, no one will trust your grid again (trust is not a coin that is easily minted). On the other hand, if you require that every programmer get a certificate (as well as an account) you make participating in the grid that much more difficult. Thus, there is a trade-off of security versus usability.

At present, the question of how to best handle the authentication of grid programmers remains open. It is one thing to sign applications for students or employees. It is quite another to sign applications for people you have never met. The former case is risky; the later case is grievously unsafe.

VI. How do you get a certificate?

The steps needed to obtain a free personal e-mail certificate from Thawte start with a visit to https://www.thawte.com/email/index.html#. A free certificate will not show your name when you sign your jar files. JAWS, for example, will show your name as: "Thawte Freemail Member". A new certificate will cause the web start program, launched from the screen saver, to pause and ask if the certificate is trusted. The certificate identifies you as a generic "Thawte Freemail Member". If you want the certificate to identify you, by name, then the grid programmers needs to join the "Freemail Web of Trust" membership, described in the following section.

Thawte has issued a new API available to certificate resellers (like DocJava, Inc.). This new API holds the promise of further enabling the automation of certificate signing and issuance. The question of how to make use of the new API remains open.

VII. Freemail Web of Trust

To obtain a certificate that authenticates your identity (by using the best efforts of the CA (Certificate Authority), the grid programmer joins the Web Of Trust (WOT). There are two ways in which to join the WOT, the free way and the non-free way. The free way requires that you meet with "Web of Trust" notaries. They will require two forms of identification and will vouch that you are who you say you are.

The second (non-free) way to join the WOT is to obtain a "Remote Authentication" (RA). RA requires you to meet with two trusted third parties. Examples include a bank manager, a practicing attorney or a CPA. A bank manager must be the manager of a branch of a registered banking institution

There is a $25 fee for using RA and, once authorized, you become a notary yourself. We elected to pay the fee and went to two different notaries at two different local banks.

The delay in getting the verification process done is about 6 weeks. This is a lot of overhead to ask of the grid programmers.

VIII. Getting your new certificate

Downloading the new certificate requires a manual procedure that includes a visit to the CA web page. The

Initium keywizard has a means of automating parts of the process, including a method for importing certificates into the key store.

A toolkit called the *KeyUtils* class has a *cleanThawtes* procedure that automates the processing of certitificates returned by Thawte [Dallaway][4]. As a part of the Initium keywizard, *cleanThawtes* was wrappered with a GUI.

Key management is tedious and error-prone. The Initium key wizard handles a series of different situations. For example, when no "*.keystore"* file appears in the users' home, the program will prompt you for a key store file. If you do not have one, the program will offer to make one for you, adding a self-signed certificate. The program also offers to create a certificate request file, thus easing key management, a little.

The question of how to make the key management easier remains open. The idea of allowing users with accounts to submit unsigned jobs to the server is gaining increased popularity. Perhaps if they purchase access to the grid with a credit card, they are trustworthy enough.

IX. Signing Jar Files

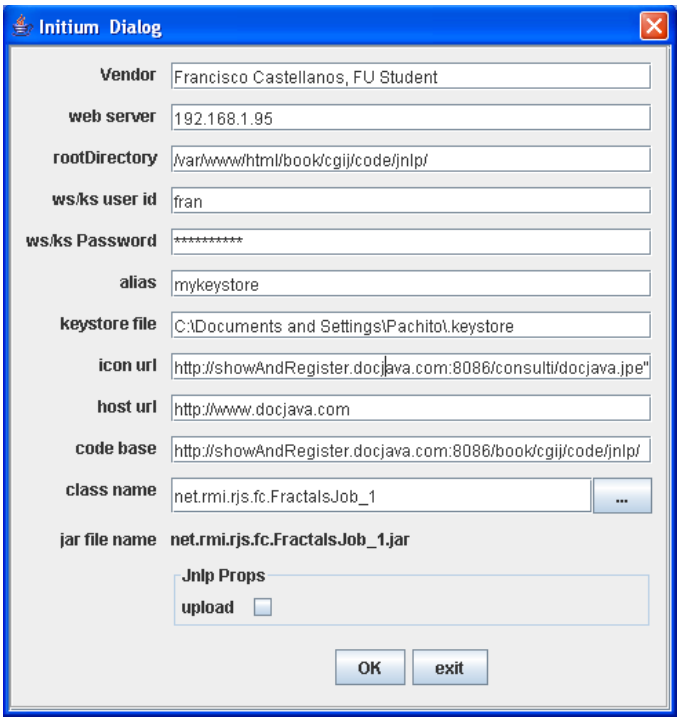
Initium programmatically signs jar files for the grid programmer. This is a far more difficult problem than it would first appear. There has been some excellent work on the programmatic signing of jar files. Scott Oaks has some code for signing jar files [Oaks 2001][5]. However, it is not compliant with 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 is public [Oaks 2004][6].

Raffi Krikorian has an excellent article on signing jar files programmatically, however, it has several problems with the code [Krikorian][7]. 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. Contacting the author did not yield a bug fix. As a result, the Initium wizard makes used of the *sun.util.Jarsigner* for jar file signing.

X. JNLP Synthesis and Deployment

The Initium wizard generates a Java Network Launch Protocol (JNLP) file. A JNLP file contains several parameters, including the file name, path name, class name, resource requirements, title, vendor, homepage, etc. 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.

Initium uses secure copy protocol (SCP) to upload the JNLP file (along with the signed jar) to a web server. This enables deployment from anywhere on the Internet, provided the SCP port is open.



a) Fig. 2. Initium Dialog

Fig. 2 shows an image of the Initium dialog, prompting the user for various parameters. Most of the parameters remain the same from one run to the next. Thus, they are stored in the user preferences, so that their entry is retained. All the parameters needed to deploy the JNLP file, sign the jar file, and perform the SCP upload are entered via the Initium dialog. Other dialogs present when class path issues arise.

XI. Screen-Savers

This section describes Java-based screen-savers for Microsoft Windows, Linux and MacOSX, for enabling a Java-based grid-computing environment.

A screen-saver is a program that activates during a period of user-computer quiescence. Detection of this quiet time enables the use of otherwise wasted CPU cycles. When the period of user-computer quiescence ceases, the screen saver terminates any compute jobs and releases the computer back for general use. Such a program constitutes a first step toward utilizing otherwise idle compute resources in a grid computing system.

We are motivated to study screen-savers because they represent a minimally-invasive technology for volunteering compute servers. Typically, utilization occurs between 40 and 60 hours out of a 168-hour week. This represents approximately 35% utilization. Our theory is that a screen-saver based *cycle scavenging* will improve this number dramatically.

We are also motivated to provide this in a Java-based environment in order to capitalize on Java's ability to

execute the same program on many different platforms. This makes a larger universe of grid-compute servers available without requiring changes to the computational program.

In order to implement a screen-saver, we implemented a *renderFrame* method that is called by the screen-saver framework [JDIC][8].

The alternative to creating a Macintosh-based screen-saver is to run X-windows under the Macintosh. Our general feeling was that this is an atypical use of the Macintosh, and we preferred a solution that makes use of the native window manager (quartz) for the Macintosh. The code for writing screen-savers is widely known [Christensen][9]. Our contribution is in the integration of the screen-savers with the CS.

An example of a simple screen-saver follows:

```
Package
  org.jdesktop.jdic.screensaver.bouncing
  line;

  public class BouncingLine  extends
  SimpleScreensaver {
  public void init(){…}
  public void paint( Graphics g ) {…}
  public void destroy() {…}
  …
}
```

In the *BouncingLine* class, the *paint* method erases the previous painted line and draws the new line. For example:

```
public void paint( Graphics g ) {
        Component c =
  getContext().getComponent();
        int width = c.getWidth();
        int height = c.getHeight();

        // Erase old line:
        g.setColor( c.getBackground() );
        g.drawLine( p1.x,
            p1.y, p2.x, p2.y );

        // Move points and
        // bounce off walls:
        bounce( p1, dir1,
          width, height );
        bounce( p2, dir2,
          width, height );

        // Draw new line:
        g.setColor( lineColor );
        g.drawLine( p1.x, p1.y,
          p2.x, p2.y );
    }
```

Two other callback methods include *init and destroy*, invoked when the screen-saver starts and stops. The *init* method is invoked upon screen-saver start up. An example implementation follows:

```
public void init(){
   ScreensaverSettings settings =
    getContext().getSettings();
  Component c =
    getContext().getComponent();
  int width = c.getWidth();
  int height = c.getHeight();
  randomizePoint( p1, width, height );
  randomizePoint( p2, width, height );
  dir1 = new Point(
       randomVector(), randomVector() );
  dir2 = new Point(
        randomVector(), randomVector() );
  String colorOption =
        settings.getProperty( "color" );
```

We have created a web start method for automatically deploying and installing the screen-saver. The screen saver is *beamed over* via the web start application and place in the proper location for user screen-savers (e.g., on a Mac, this is ~/Library/Screensavers/). We detect the operating system and CPU type, before the transfer. We have routines for determining if the machine is a PPC Mac, Intel Mac, x86 Linux or MS Windows. Support for multiple machines is both tedious and time-consuming. The web start application launched by the screen-saver framework is a compute server. To demonstrate how error-prone such code can be, consider that MS Windows stores its screen-saver in a different place for *each* minor version release. For an example of how this influences the code, consider the following:

```
public static String
  getScreenSaverHome() {
        final String system32 =
  "c:\\windows\\system32";
        final String winNt =
  "c:\\winnt\\system";
        final String system =
  "c:\\windows\\system";
        if (OsUtils.isWindowsXp())
  return system32;
        if (OsUtils.isWindowsNt())
  return winNt;
        if (OsUtils.isWindows2000())
  return winNt;
        if (OsUtils.isWindows98())
  return system;
        if (OsUtils.isWindows95())
  return system;
        System.out.println("er!:could
  not identify OS:" +
  OsUtils.getOsName());
        return system32;
}
```

Thus, the windows home for Windows NT is different from XP and Windows 98. Naturally, there are different locations for Unix and Mac OS too. In short, there are no standard locations for screen-savers.

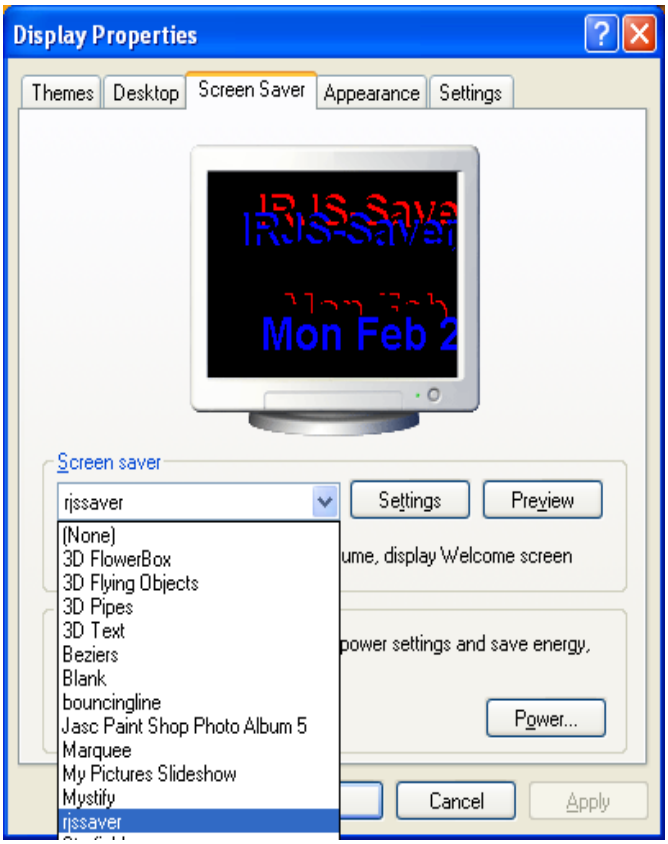


Fig. 3. RJS Screen-Saver

After installation, the user selects the new screen-saver, just like any other, as shown in Fig. 3. After determining the location of the WebStart client application on the users' machine the screen-saver uses a URL to start the job via an invocation to the *Runtime*. These efforts are detailed in [Lyon et Al. 2006(a,b,c)][10][11][12].

Constructing a native-method framework for controlling web-start applications from a screen-saver was both painful and educational. Screen-savers are surprisingly difficult to design, correctly. They consume a great deal of grid development time and require climbing a steep learning curve that encompasses multiple languages and platforms. Further, development and support of these programs requires extensive cross-platform testing. It is little wonder that the Java grid computing community has almost totally ignored screen-savers, before now.

XII. Middleware

This section describes the middleware needed to deploy jobs to non-geographically co-located clusters with decentralized look-up severs.

A Web Server (WS) has a Java Network Launch Protocol (JNLP) file that references a signed jar file. The *computation jar* contains a job for the computation server. The LUS sends back answers to the server using RMI over SSL (RMI/SSL). When a CS starts, it registers with a LUS. The LUS then updates its list of computation servers in the cluster. Look-up servers start via Java Web Start. The LUS, then proceeds to delegate tasks to new registered CS. Fig. 4 depicts the events mentioned.

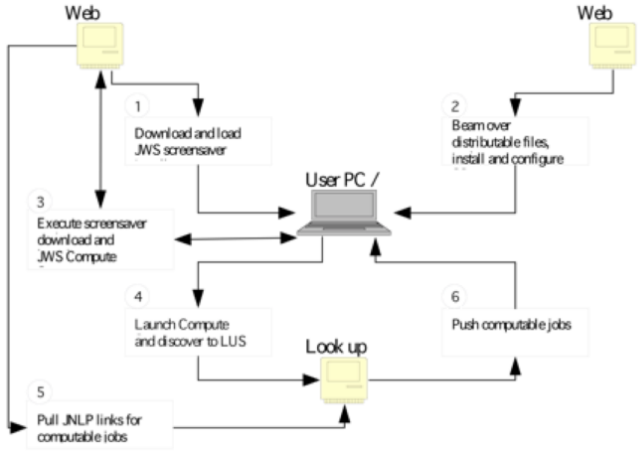


Fig. 4 IRJS System events

All LUS – CS communication is on the LAN behind the firewall. Most grid systems use SSL for authentication, but by default do not establish encrypted communication in a secure manner [Globus 2][13]. Our approach requires that we encrypt all WS-LUS communication via a session key. In order to establish the session key, WS and LUS must agree on a shared key without sending any secret data in the clear. To do this, we use the RSA key exchange algorithm for SSL key exchange. In RSA key exchange, the WS encrypts a number of random bytes with the LUS's public RSA key and they both use this shared secret to create the session keys. The Message-Digest 5 (MD5) algorithm ensures message integrity. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA [Rivest][14]. The CS receives a URL to a JNLP file and downloads the Java Web Start "job" in order to compute it. The owner of the CS must accept the job owner certificate. Once the certificate is trusted, all jobs signed by it execute automatically.

RSA algorithms ensure secure communication and the MD5 message digest ensures that no one has altered the data in transit.

An alternative approach to security might use the Kerberos network authentication protocol [Kerberos][15] or SSH protocol [OpenSSH][16]. However, such tools are vulnerable, as the attackers can gain access to user's password as it is typed [Basney][17].

RJS architecture uses Java Web Start technology and is operating system independent. Java Web Start provides the power to launch full-featured Java applications with a single click without going through complicated installation procedures [Sun 2004b][18]. The system has three major sub-systems: the Web Server, Look-up Server and Computation Server as shown in Fig. 5. Different computers can run all three systems [Lyon et Al. 2006d][19]

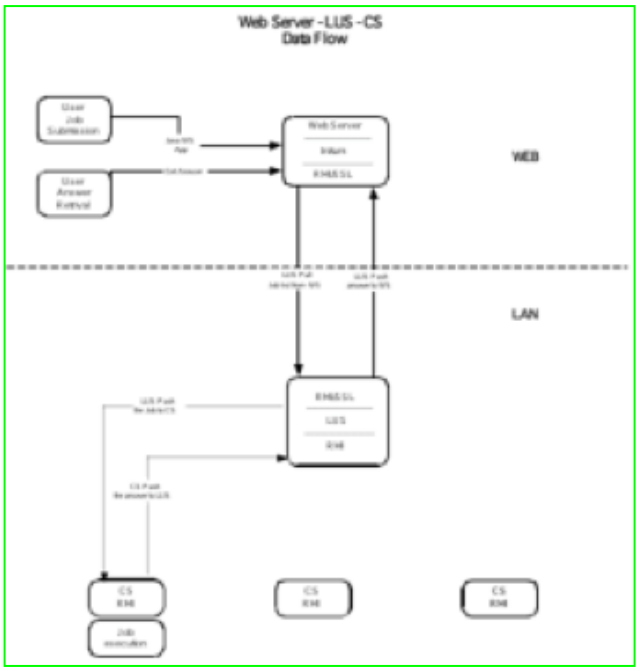


Fig. 5 RJS data flow

XIII. Related Work

Grid Computing is not new, nor for that matter is grid security. The use of grid computing on a heterogeneous network is also not new. What is new is our use of Java Web Start to distribute jobs on the grid. This opens the door to grid computing on a heterogeneous network for Java programmers.

Globus uses the Grid Security Infrastructure (GSI, Globus Project Toolkit 3.0) to implement grid security [Silva][20]. GSI provides a number of useful services for grids, including mutual authentication and single sign-on. A central concept in GSI authentication is the certificate. A certificate identifies each user of the Grid [Globus][21]. Manipulating grid certificates using Globus Toolkit 3.0 (GT3) in Windows environments is awkward. It requires the system administrator or user to install GT3 on Linux systems in order to use command line scripts to generate certificates. Users must move those certificates to their Windows system [Silva 2][22].

The Cog Kit contains the Globus Toolkit APIs. They generate user certificates and certificate requests. It can also sign certificates and create proxies [Globus2][23].

In many ways, our approach is similar to GSI, but we are using a pure Java implementation. In our judgment, the GSI technique is very secure; however, it is also cumbersome.

XSOAP and XCAT grid web services use the GSI to provide Public Key Infrastructure [XSOAP][24]. Also, EU DataGrid project‘s authentication and delegation is based on the Globus GSI, which is an extension of the Public Key Infrastructure [Cornwall][25]. XSOAP and XCAT grid web services use GSI. As a result, XSOAP and XCAT grid web services suffer from the same advantages and disadvantages of GSI.

The JPARSS (Java Parallel Secure Stream for Grid Computing) system has security features that enable authentication via a temporary X.509 certificate. Temporary certificates go to those who hold a certificate issued by a CA [Chen][26]. The advantage of this solution is one-time authentication, but still, there is a need for a CA to sign the temporary certificate. Furthermore, it requires a password to create X.509 certificate.

Grid Portal Development Kit (not supported anymore) gets its security using a few simple methods for setting the username, password and designated lifetime of the proxy [GPDK][27]. The system is not secure and abandoned.

JGrid introduced an Authentication Service (AS) that provides short-term certificates for users without their own certificate. In this case, a user can register with the AS, log in a custom way, and obtain a private key and certificate. Therefore, the AS is a CA (Certificate Authority) of the JGrid, providing short-term credentials [JGrid][28].

JGrid is based on JINI technology. JINI network technology, which includes JavaSpaces Technology [Flenner][29] and JINI-extensible remote invocation (JINI ERI), is an open architecture that enables developers to create network-centric services that are highly adaptive to change [Sun2000][30]. The approach of issuing short-term certificates is similar to JPARSS, thus it shares same advantages and disadvantages.

Condor provides support for strong authentication, encryption, integrity assurance, as well as authorization. Most of these security features are not visible to the user (one who submits jobs). They use configuration macros that run by the site [Condor][31]. Since the Condor project uses GSI for authentication [Condor2][32], it suffers from the same advantages and disadvantages of GSI.

The Gridbus Project has developed a Windows/.NET-based desktop clustering software and grid job web services to support the integration of both Windows and Unix-class resources for Grid [GRIDBUS][33]. Unlike RJS, the installation of Gridbus is time intensive. In most of the cases, compilation of the code is required. Also, before installation, many prerequisites, such as: IBM TSpace, Globus toolkit 2.4, MySQL or Microsoft .NET Framework 1.1, are required.

Screen-saver based grid computing systems are not new (http://boinc.berkeley.edu/) but their use for Java computing is [SETI][34]. Additionally, some screen-savers, like SETI, are closed systems in that others can only contribute CPU cycles (but not programs). Our system differs from SETI style grid computing in four ways:

1. RJS has binary portability (it is Java-based),
2. RJS enables others to submit jobs to the grid,
3. RJS automatically configures the CS,
4. RJS is secure.

In addition, Java-based screen savers have typically been

restricted to MS Windows and Xwindows (UNIX)-based systems. Our system extends the screen-saver technology to Macintosh-based systems [JDIC][35].

The creation of key tool API's is not new either [BouncyCastle][36]. Verisign has an API, but its' support has been withdrawn [Verisign][37]. The interesting thing about the Verisign API is that it enables users to revoke public keys at the CA (which is an excellent idea). Thawte now has a new API for this application. The new Thawte API enables the automation of Thawte interactions and is a topic of current research.

There have been articles on the signing of JAWS applications [Dallaway][38]. There have also been articles on the signing of Applets [Gallant][39] [Myer][40].

The creation of a key wizard seems like a logical evolution of the key tool. It is surprising, therefore, that keytool wizards are so few and far-between. A reason for this is that Sun has not open-sourced the JKS (Java Key Store) API. On the other hand, there are open-source versions of a JKS system [Marshall][41]. We have yet to test this system.

One notable keytool wizard is the BEA systems wizard for their Weblogic product [BEA][42]. Such systems are closed-source, expensive and only work on Weblogic.

Programmatic signing is not new [Krikorian][43]. Nor is the practice of reducing jar size via static dependency analysis [Sadun][44].

Resource bundling problems are not new [Lyon 2005a][45], nor are the issues of key management and the integration of deployment with programmatic signing and jar optimization [Lyon 2005b][46] [Lyon 2004][47]. However, the integration of key management automation with automatic deployment, and SDA is new and lowers the programmer effort needed for web deployment.

XIV. Conclusion

The Initium project has uploaded a series of applications via SCP. Uploading requires a location for the destination files. Perhaps that is not an optimal situation. Hard-coded JNLP HREFs 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.

Key-management automation facilitates application deployment. This is a primary motivation for creating the key wizard. The automation of key-tool functions is not hard, given proper error handling and GUI implementation effort. However, the key-tool API is neither public, nor open-sourced. The implementation effort required in order to provide the key-tool function points is substantial.

Even more disconcerting is the general lack of programmatic support for the retrieval of trusted certificates. The Thawte API may be of help here. Meanwhile, the user must still interact with a CA in order to go through a process of getting a certificate. This limitation is less a technical one than a security one. Establishing in-house notaries will speed registration when there are several geographically co-located programmers present.

Screen-saver computing, in Java, remains an immature technology. Ports of the SaverBeans development kit are still not available for the Mac. Our own solution for the Mac is probably sufficient, for now, however, a unified framework for all platforms would be preferable.

In summary, grid computing still needs help in providing a one-stop shop that integrates security, deployment and proper packaging. A series of wizards can help ease the deployment and packaging burden. However, there are still deeper security policies that need to be addressed before grids can be open global resources.

XV. Future Work

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: The class resides in the *sun.util* package, and this package is not generally stable (nor even endorsed for general use!). Further, the *sun.util.JarSigner* is intended to be used from the command line. The verification mechanism implemented in the JarSigner terminates the callers' thread of execution, an unwelcome side effect. With JDK1.5, the *sun.util.JarSigner* API moved into *tools.jar* [Sun 2005b][48]. A stable, public *JarSigner* API would help automate the signing task.

Certificate management is still complex and impoverished. If the trust is misplaced, it should be possible to check a list of *bad key risks*. A standard for certificate revocation has yet to be formulated (what if it is lost, or stolen?).

Any entity that signs the code of others is at risk. Thus, it seems best to issue keys to individuals and to develop a web of distrust (WOD). If enough notaries in the WOT indicate that an individual is a bad risk (con-artist, virus author, etc.) it should be possible to revoke their key. The question of how to implement this, or even how to check a database of revoked keys is open. It may be up to the CA to put this into the API.

The question of how to automate the discovery of the existence of a firewall, and a proxy server, remains open. This is critical to the correct configuration of JAWS.

The question of how to scan the Java byte codes for symbolic references is open. This is important to the correct operation of the SDA pre-processing feature of the Initium system. Even with a byte code scanner, SDA can fail to work properly.

Dynamic dependency analysis (DDA) is an obvious next step. The question of how to implement DDA remains open. One possible answer might be to log loaded classes from a modified class loader, at run-time.

A better class-path management scheme needed to facilitate robust SDA operation. At present, the GUI prompts the user for the location of classes that it is not able to resolve during the SDA phase.

One of the open problems that remain with JAWS is the set-up problem. Manually setting the proxy web server 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 Windows. To add insult to injury, Windows requires a reboot after the installation (at least under Windows Professional).

JXTA may help address the proxy issue, but other installation requirements (i.e., system-admin privilege, remain).

We are working to extend the ideas presented in the paper to help with clusters and grid computing. We have already deployed prototype RMI applications via the Initium system and have a full-fledged grid computing system in the works. The question of how effective this system will be in automating grid computing remains open.

Finally, the installation of screen-savers on an MS Windows system that is locked-down remains open. If we lack write access to the *Windows32* directory, then there may be no way to add a screen-saver.

An added complication to the research is that Oracle has discontinued support for Java Webstart in Java 11, however they have encouraged the use of jlink and/or third part packaging and deployment solutions, which is presently a topic of current research [49].

ACKNOWLEDGMENT

This research was made possible, in part, by a Faculty Research Grant from Fairfield University.

REFERENCES

[1] [Lyon 2005c] "Project Initium: Programmatic Deployment" by Douglas A. Lyon, Journal of Object Technology, vol. 3, no. 8, September-October 2004, pp. 55-69. http://show.docjava.com:8086/pub/document/jot/web.pdf

[2] [Sun 2004a] "Networking Properties" by Sun Microsystems http://java.sun.com/j2se/1.4.2/docs/guide/net/properties.html

[3] [Chen] "Java Parallel Secure Stream for Grid Computing" by Jie Chan, Walt Akers, Ying Chen and William Watson III, High Performance Computing Group, http://www.jlab.org/hpc/papers/jparss.pdf

[4] [Dallaway] Richard Dallaway, "Java Web Start and Code Signing", May 2002 http://www.dallaway.com/acad/webstart/

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

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

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

[8] [JDIC] Java.net: "JDIC project home", https://jdic.dev.java.net/

[9] [Christensen] "Writing a Screen Saver Module" by Brian Christensen, April 10, 2001, http://www.cocoadevcentral.com/articles/000011.php

[10] [Lyon et Al. 2006a] "Initium RJS: Screensaver in Java, Part 1, MS Windows" by Douglas A. Lyon and Francisco Catellanos, submitted to the Journal of Object Technology, vol. 5, no. 4, May-June 2006, pp. 7-16.

[11] [Lyon et Al. 2006b] "The Initium RJS Screensaver: Part2, UNIX" by Douglas A. Lyon and Francisco Castellanos, in Journal of Object Technology, vol. 5, no. 6, July-August 2006, pp. 7-15.

[12] [Lyon et Al. 2006c] "A Macintosh Screensaver in Java : Part3",by Douglas A. Lyon Pawel Krepsztul and Francisco Castellanos, in Journal of Object Technology, vol. 5, no. 7, September-October 2006, pp. 9-17.

[13] [Globus2] Overview of the Grid Security Infrastructure (GSI),http://www-unix.globus.org/security/overview.html

[14] [Rivest] "MD5 Algorithm" by Ronald L. Rivest, April 1992, http://www.kleinschmidt.com/edi/md5.htm

[15] [Kerberos] "Kerberos: The Network Authentication Protocol", http://web.mit.edu/kerberos/www/#what_is

[16] [OpenSSH] OpenSSH, http://www.openssh.org

[17] [Basney] "A Roadmap for Integration of Grid Security with one time Passwords" by Jim Basney, Von Welch, Frank Siebenlist, April 18, 2004, http://www.ncsa.uiuc.edu/~jbasney/grid-otp.pdf

[18] [Sun 2004b] "Java Secure Socket Extension (JSSE)" by Sun Microsystems, http://java.sun.com/products/jsse

[19] [Lyon et Al. 2006d] "Heterogeneous Autonomic Screen-Saver CPU Scavenging", by Douglas A. Lyon, Pawel Krepsztul, New England ASEE Conference, March 17-18th, Worcester, MA 2006.

[20] [Silva] "Manage X.509 certificates in your grid with Java Certificate Services" by Vladimir Silva, ibm.com, http://www-106.ibm.com/developerworks/grid/library/gr-jsc/, October 2003

[21] [Globus] The Globus Alliance, http://www.globus.org

[22] [Silva 2] "Using Java technology with Globus Grid Security Infrastructure" by Vladimir Silva, ibm.com, http://www-106.ibm.com/developerworks/library/gr-ggsi/, September 2003.

[23] [Globus2] Overview of the Grid Security Infrastructure (GSI),http://www-unix.globus.org/security/overview.html

[24] [XSOAP] "Grid Web Services: Security in XSOAP and XCAT", http://www.extreme.indiana.edu/xgws/security/

[25] [Cornwall] "Security in multi-domain Grid Environments" by Linda Cornwall and team, Kluwer Academic Publisher, April 15,2004 http://www.urec.cnrs.fr/publications/GridJournal-EDG-SCG-paper.pdf

[26] [Chen] "Java Parallel Secure Stream for Grid Computing" by Jie Chan, Walt Akers, Ying Chen and William Watson III, High Performance Computing Group, http://www.jlab.org/hpc/papers/jparss.pdf

[27] [GPDK] "Grid Portal Development Kit", DOE Science Grid, http://www.doesciencegrid.org/gridportal.html

[28] [JGrid] "The security architecture of the JGrid System" by Mark Magyarodi

<http://pds.irt.vein.hu/documentation/JGrid_security.pdf>

[29] [Flenner] "First Contact: Is There Life in JavaSpace" by Robert Flenner, http://www.onjava.com/lpt/a/750

[30] [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

[31] [Condor] 3.7.3 Security Configuration, http://www.cs.wisc.edu/condor/manual/v6.4/3_7Security_In.html

[32] [Condor2] "Condor and the grid" by Karthik Ram Venkataramani, www.ccr.buffalo.edu/grid/ download/ condor -and-the-grid.pdf

[33] [GRIDBUS] "The Gridbus Middleware 2004 ", http://www.gridbus.com/middleware/

[34] [SETI] Seti@home, <http://setiathome.ssl.berkeley.edu/>

[35] [JDIC] Java.net: "JDIC project home", https://jdic.dev.java.net/

[36] [BouncyCastle] http://www.bouncycastle.org/specifications.html, An open-source API that provides JCE features.

[37] [Versign] "Trust Services Integration Kit 1.7" http://home.postech.ac.kr/~chokee/cs499/tsik-1.7/tsik-1.7/api/com/verisign/xkms/tools/XKMSKeyStore.html, and http://www.verisign.com/static/005314.pdf

[38] [Dallaway] Richard Dallaway, "Java Web Start and Code Signing", May 2002 http://www.dallaway.com/acad/webstart/

[39] [Gallant] Michel I. Gallant, "Thawte Code Signing Certs with Authenticode, Netscape and Sun Signing", http://www.jensign.com/JavaScience/Thawte/

[40] [Myer] "Grid watch: GGF and grid security" by Thomas Myer, 13 May 2004,<http://www-106.ibm.com/developerworks/library/gr-watch4.html>

[41] [Marshall] Casey Marshall, "An implementation of the JKS key store" http://metastatic.org/source/JKS.html

[42] [Bea] "The Bea Systems Keytool Wizard" http://edocs.bea.com/wli/docs70/b2bsecur/keystore.htm

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

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

[45] [Lyon 2005a] "Resource Bundling for Distributed Computing" by Douglas A. Lyon, Journal of Object Technology, vol. 4, no. 1, January-February 2005, pp. 45-58.

[46] [Lyon 2005b] "The Initium X.509 Certificate Wizard" by Douglas A. Lyon, Journal of Object Technology, vol. 3, no. 10, November-December 2004, pp. 75-88.

[47] [Lyon 2004] "The Initium X.509 Certificate Wizard" by Douglas Lyon, private communication with the author, November 2004.

http://show.docjava.com:8086/pub/document/jot/initium.pdf

[48] [Sun 2005b] Private correspondence with Vincent Ryan <Vincent.Ryan@Sun.COM>.

[49] [Oracle] Java Client Roadmap Update, March 2018. https://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (*Java, Digital Signal Processing, Image Processing in Java* and *Java for Programmers*). He has authored over 50 journal publications. Web: http://www.DocJava.com.

Pawel Krepsztul. Earned his Master's Degree in the Electrical and Computer Engineering from the Fairfield University in August 2005. His research interests include grid computing. Currently he is employed by Pepsi Bottling Group in Somers, NY as a software developer.

Francisco Catellanos. Earned his B.S. (Hons) degree in computer science at Western Connecticut State University. Francisco Castellanos worked at Pepsi Bottling Group in Somers, NY as a software developer. Currently he is working on a thesis to complete his M.S. degree in Electrical and Computer Engineering from Fairfield University. His research interests include grid computing. He is currently employed by Access Worldwide in Boca Raton, FL as a software developer.