JOURNAL OF OBJECT TECHNOLOGY

. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6. No. 1, January-February 2007

The Saverbeans Screensaver and Initium RJS System Integration: Part 5

Douglas Lyon and **Francisco Castellanos**

Abstract

This paper describes the integration of our Java-based screensaver framework with our Initium Remote Job Submission (IRJS) grid computing middleware. Initium RJS is a Java Web Start (JAWS) based grid-computing technology. This is part 5 of a 5 part series. In parts 1, 2, and 3, we described screensavers for MS Windows, Xwindows, and the Macintosh. Part 4, describe automatic deployment of the screensavers into the MS Windows and XWindows systems.

This paper introduces the use of our grid system. We provide an example that shows how to take a von Neumann style program, break it up, and deploy it to our grid system.

1 INTRODUCTION

This article describes the process followed to integrate the IRJS screensaver with the IRJS grid computing middleware. Our goal is to provide a minimally invasive CPU scavenging technology. The IRJS screensaver launches a Compute Server (CS) when the computer enters a quiescent state. The CS joins the grid and volunteers its resources. The IRJS screensaver terminates the CS when any user input is detected. We use the [Saverbeans] framework to create our screensaver and to allow such behavior.

The primary goal of this paper is to describe an example of the transformation of a von Neumann style program into a concurrent program that makes use of our grid framework. We shall confine ourselves to a simple first example that makes use of a well-known fractal computation called the Mandelbrot set. We also summarize the basis theory of operation of our grid framework.

2 INTEGRATION OF A SAVERBEANS SCREENSAVER AND A GRID SYSTEM

We use a Compute Server (CS) that uses multicasting to discover a Lookup Server (LUS) over a local network. The CS announces that it is available, and provides benchmark data

Cite as follows: Douglas A. Lyon and Francisco Castellanos “The Saverbeans Screensaver and Initium RJS System Integration: Part 5”, in *Journal of Object Technology*, vol. 6. no. 1, January-February 2007, pp. 35-57

to the LUS. The benchmark data describes computer resources, and it is used by the LUS to allocate tasks to the CS. The LUS holds a pool of previously partitioned jobs that need to be processed, and it allocates these jobs to available resources. The main task of the CS is to process or compute the job and to transmit the results back to the LUS [Pawel and Lyon]. The role of the screensaver is to detect user-computer quiescence and use this interval to volunteer CPU cycles to the LUS.

The *SaverBeans* framework provides two methods that can be shadowed to alter the behavior of screensaver initiation and termination. These two methods are *init()* and *destroy()*, and they are defined in the abstract class *ScreensaverBase*. We subclass the `SimpleScreensaver` in order to create our own screensaver. The *init()* method is called during the screensaver startup. In our Java class IRJS Saver we have implemented the *init()* method to not only initiate the state of our screensaver, but also to invoke the CS as shown in Example 2-1.

Example 2-1

```
public void init() {
        ScreensaverSettings settings = getContext().getSettings();
        Component c = getContext().getComponent();
        int width = c.getWidth();
        int height = c.getHeight();
        randomizePoint( p1, width, height );
   ….

       /*Initiate Compute Server and Monitor once.
     Init method is invoked more than once during the execution the
screensaver.
      */the variable iCount is static.

iCount = iCount + 1;
        if (iCount < 2){
         startComputeServer();
         launchLogMonitor();
        }
    }
```

The *startComputeServer()* method called in Example 2-1 invokes the Compute Server application using Java Web Start as shown in Example 2-2**.** Notice the parameters passed to the ***exec*** method in the Runtime object ***rt.*** They are the application name “javaws” (Java Web Start), the command ***–Xnosplash*** to avoid any splash screens, and the location or URL of the CS.

Example 2-2

```
private void startComputeServer(){
    Runtime rt;
   Process p;

  String[] params = {"javaws", "-Xnosplash",
"http://www.myjavaserver.com/~fsophisco/" +
               "net.rmi.pawelGrid.LusCs.CsMain.jnlp"};

               rt = Runtime.getRuntime();
         try{
               p = rt.exec(params);
               p.waitFor();

         }catch(Exception e){
               System.out.println("Error @ startComputeServer()");
               e.printStackTrace();
         }
 }
```

Example 2-1 shows a call to the *launchLogMonitor()* method and this is listed in Example 2-3**.** It has the purpose of creating a thread that periodically monitors and reads from the CS log file to find the state of the CS. We use the state of the CS to help create a display in the screensaver frame, and it occurs in the shadowed *paint()* method, which is discussed later in this section.

Example 2-3

```
     private void launchLogMonitor(){

          Thread t =  new Thread(new Runnable(){
                      public void run(){
                      try{
                            while (true){
                                  Thread.sleep(5000);
                                  readCSLog();
                            }
                      }catch(Exception e){
                            e.printStackTrace();
                      }
                      }
                      //Read CS status from Log. Set message to be
displayed.
                      public synchronized void readCSLog(){
                      try{
                            fout = new RandomAccessFile(cslogFile,
"rw");
                                 message = fout.readLine();
                                 if (ms_length != message.length())
                                 {
                                       ms_length = message.length();
```

```
                                        new_ms = true;
                                    }
                                    fout.close();
                                }catch(Exception e){
                                    e.printStackTrace();
                                }
                                }
            });
            t.start();
        }
```

3 DETECTION OF USER INPUT AND TERMINATION OF COMPUTE SERVER

The CS has been built with the capability of monitoring and detecting termination messages from an external application. In the case of receiving a termination message, the CS proceeds to perform any cleanup and communicates with the LUS, and finally terminates execution. This process restores the CS to its initial state when the user returns to use his/her computer.

Subclasses of `SimpleScreensaver` can optionally implement the *destroy()* method to perform any cleanup before the screensaver is destroyed. In our case we want to communicate with the Compute Server to inform it that user input has been detected and that it needs to terminate itself. We accomplish this by creating a directory in a common place that the CS monitors periodically, as shown in Example 3-1**.**

Example 3-1

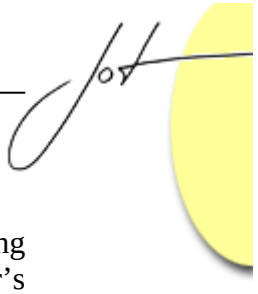
```
private static String fileSep = System.getProperty("file.separator");
private static String tmpDir = System.getProperty("java.io.tmpdir");
public final static File killFile = new File(tmpDir +
                                    fileSep +
                                    "killcs");

protected void destroy(){

          cal= Calendar.getInstance();
          killFile.mkdir();
          System.out.println("CS Stoping at "+
cal.getTime().toString());
     }
```

4 PAINTING THE NEXT FRAME

We have shadowed the method *paint()* to enable a screen display. The method *LaunchLogMonitor()*, shown in Example 2-3 starts a monitor to periodically obtain the status of the computer and to place it, in a string format, into an instance variable. For the purpose of observing the status of the Compute Server in the screen, we have included



the string message obtained from the monitor as shown in Example 4-1**.** The String simply bounces against the walls, and it is changed every time the Compute Server's status changes as shown in Figure 4-1 and Figure 4-2.

Example 4-1

```
public void paint( Graphics g ) {

         Component c = getContext().getComponent();
        int width = c.getWidth();
        int height = c.getHeight();
        Point c_point = new Point();

        //Write SS Name
        g.setFont(new Font("Arial", Font.BOLD , 25));
        g.setColor(Color.red);
        g.drawString(ssName, 30, 30);

      ….
        g.setFont(new Font("Arial", Font.BOLD , 25));
        g.drawString(message, p1.x, p1.y);

        // Erase old drawings:
        Point pe = points[indx_b];
        if (pe != null)
        {
           g.setColor( c.getBackground() );
           g.setFont(new Font("Arial", Font.BOLD , 25));
           g.drawString(message, pe.x, pe.y);
        }

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

        //Draw String
        g.setColor(Color.blue);
        g.setFont(new Font("Arial", Font.BOLD , 25));
        g.drawString(message, p1.x, p1.y);
        ….
    }
```

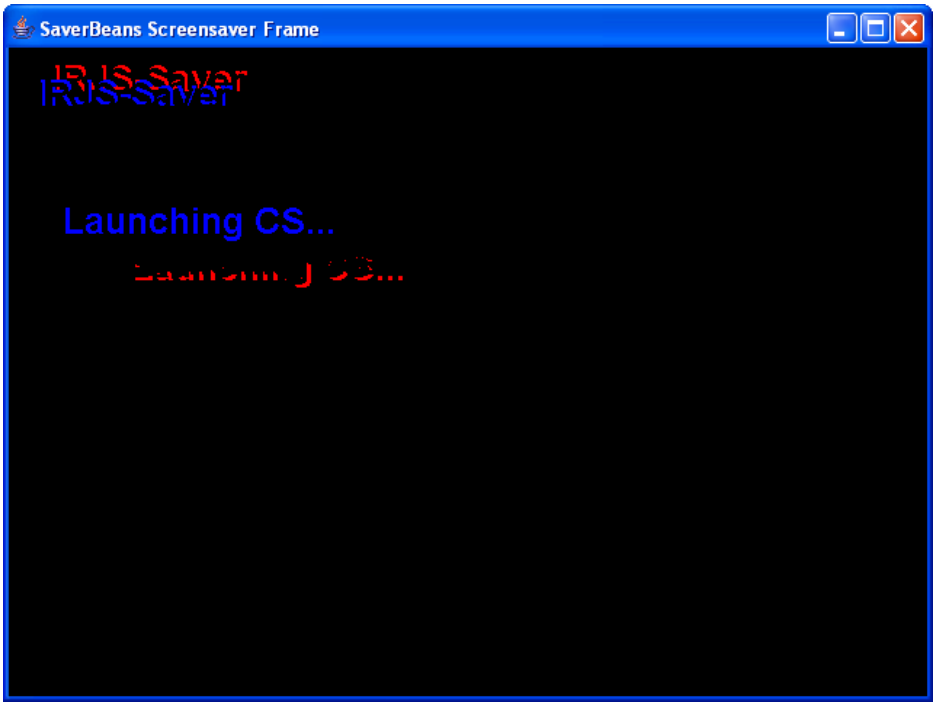


Figure 4-1 IRJS Screensaver: Compute Server is launching

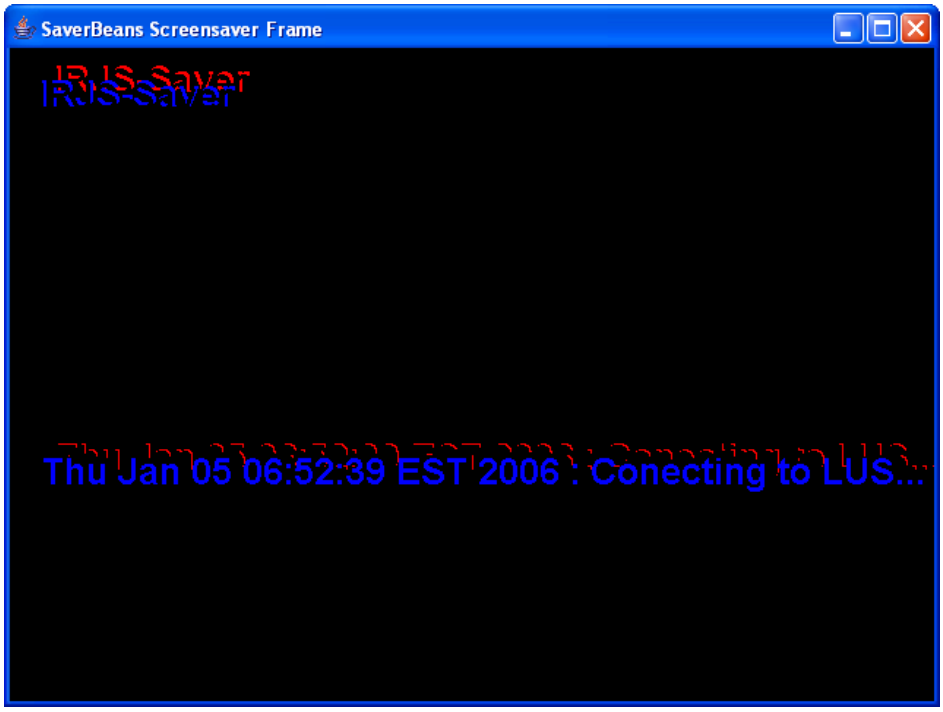


Figure 4-2 IRJS Screensaver: CS is connecting to LUS

5 SETTING YOUR OWN SAVERBEANS SCREENSAVER PROJECT

The following steps describe how to create your own screensaver project. For details about requirements, and building and compiling the project, please refer to [Lyon and Castellanos].

1. Make a copy of the startup directory located in the *SaverBeans* home, as shown in Figure 5-1.

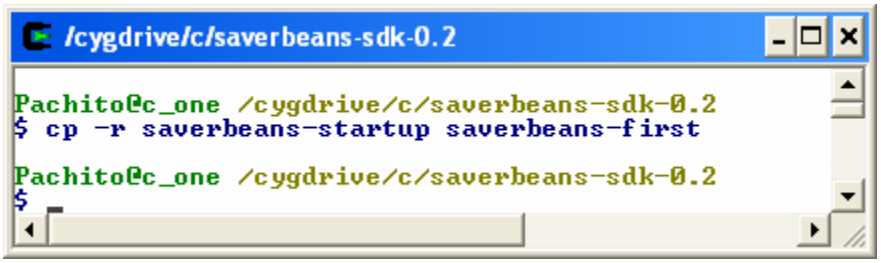


Figure 5-1. Creating a copy of Saverbeans Startup project.

2. The file *building.properties.sample* inside the rename startup directory should be copied to *build.properties.* This file contains the one important property, the location of the SaverBeans sdk home, and it should be set to the appropriate location.

```
# See http://jdic.dev.java.net/
saverbeans.path=C:/saverbeans-sdk-0.2
```

3. Edit build.xml. Replace the *bouncingline* names and locations with new names as shown in Figure 5-2. For example, change:

```
<property name="screensaver" value="bouncingline" />
 <property name="screensaver.class"

value="org.jdesktop.jdic.screensaver.bouncingline.BouncingLine" />
To :
<property name="screensaver" value="rjssaver" />
    <property name="screensaver.class"
        value="org.jdesktop.jdic.screensaver.rjssaver.RjsSaver" />
```

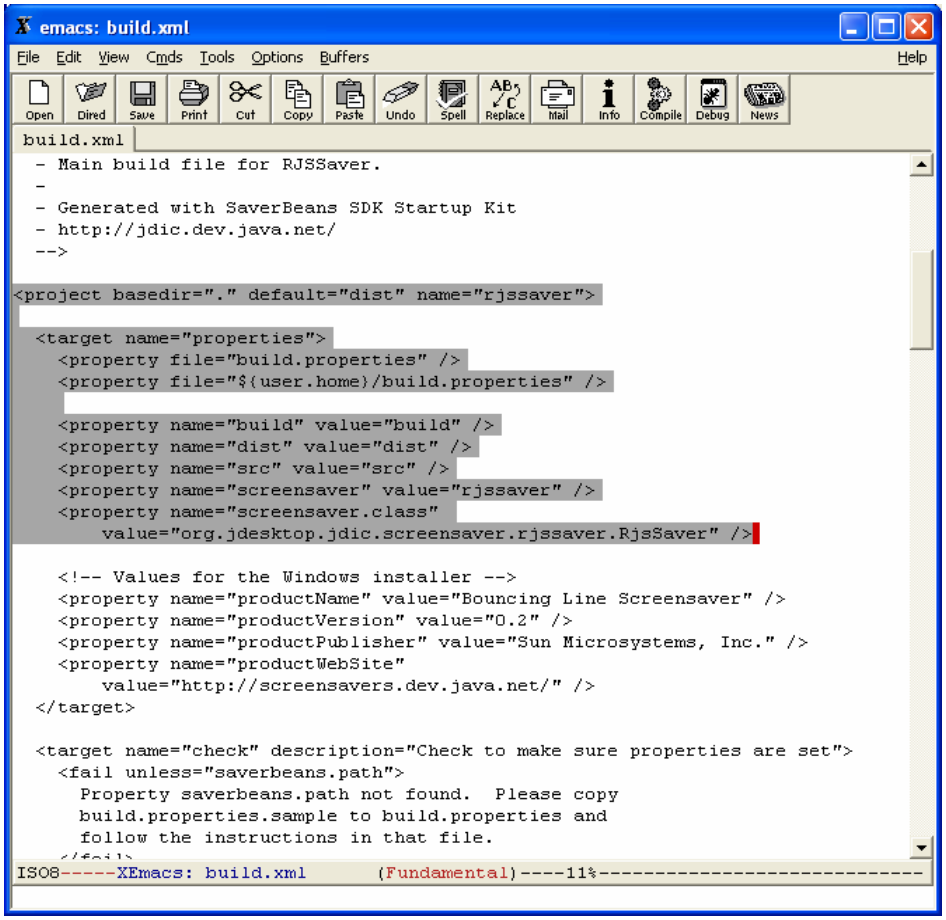
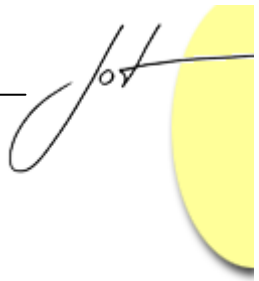


Figure 5-2 Contents of build.xml File

4. Rename your directories and files to match with the ones edited in the build.xml file. Most files and directories exist within the src folder. Below are the files and directories to be renamed:
    4.1. Rename the configuration file *bouncingline.xml* located at *src/config*
    4.2. Rename the directory bouncingline located at *src/java/org/jdesktop/jdic/screeensaver*.
    4.3. Rename the java class *bouncingline.java* located at *src/java/org/jdesktop/jdic/screeensaver/<new name>/bouncingline.java*. Remember to change the name of the package and class name inside the file as shown in Example 5-1.



Example 5-1

```
package org.jdesktop.jdic.screensaver.rjssaver;

import org.jdesktop.jdic.screensaver.SimpleScreensaver;
….

public class RjsSaver
    extends SimpleScreensaver
{
```

4.4. Lastly before making any modifications to the java class or anything else. Run *ant debug* command to make sure that the project is working correctly as shown in Figure 5-3. The screensaver should launch inside a frame.

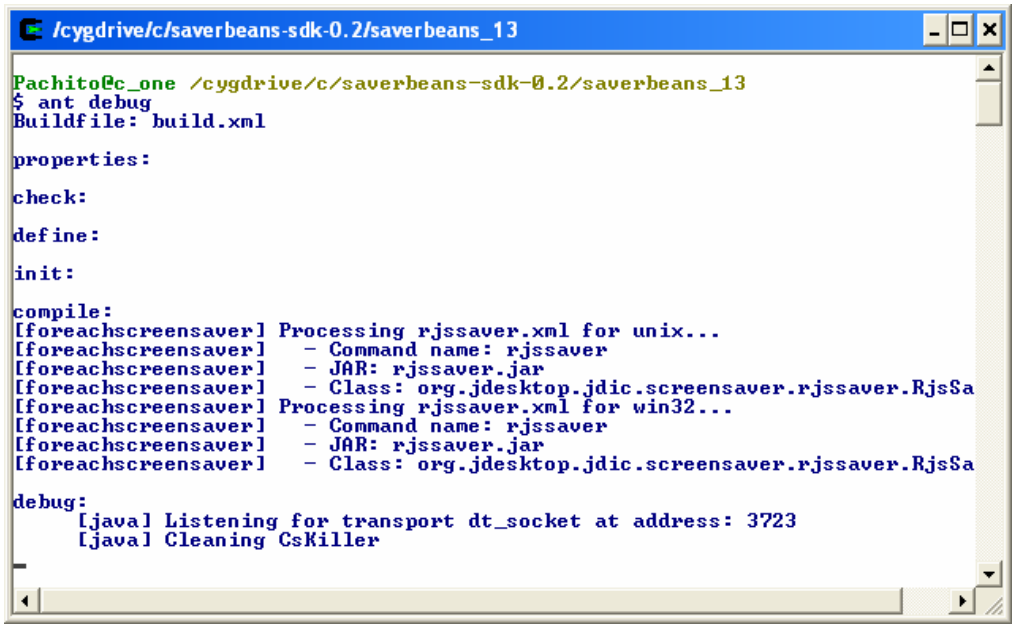


Figure 5-3. Shows the output after executing ant debug.

6 PARTITIONING A VON NEUMANN-STYLE SAMPLE PROGRAM

To demonstrate the use of the IRJS System, computable jobs must be submitted to it. In this section, our goal is to take a von Neumann style program and to partition it into executable pieces. The initial sample program produces an image of the Mandelbrot set. The main goal is to process an image using the Mandelbrot set algorithms, and to display the output image into a frame as shown in Example 6-1.

Example 6-1

```
public void testMandelbrot() {
        final Dimension dim = new Dimension(400, 400);
        ClosableJFrame cjf = new ClosableJFrame("mandlebrot") {
            public void paint(Graphics g) {
                Dimension d = getSize();
                Image i = getMandelbrot(d.width, d.height);
                g.drawImage(i, 0, 0, null);
            }
        };
        cjf.setSize(dim.width, dim.height);
        cjf.setVisible(true);
}
```

The *getMandelbrot()* method, called in Example 6-1, takes two parameters, the width and height of the image to be processed. It uses these parameters to create an image bean, where all the data is stored. It continues to call the *Mandelbrot()* method and passes the RGB arrays from the image bean. Example 6-2 shows *getMandelbrot()* method.

Example 6-2

```
public Image getMandelbrot(int w, int h)
{
ShortImageBean sib = new ShortImageBean(w, h);
      mandelbrot(sib.getR(), sib.getG(), sib.getB());
      return sib.getImage();
}
```

The *Mandelbrot()* method applies the Mandelbrot set algorithms and generates data into the RBG bean arrays as shown in Example 6-3.

Example 6-3

```
public void mandelbrot(short[][] r, short[][] g, short[][] b) {
  int height = r[0].length;
        int width = r.length;
        int Clr;
        float pixelr, pixeli;
        for (int y = 0; y < height; y++)
            for (int x = 0; x < width; x++) {
                pixelr
                        = mandleBrotDimensions.getxMin() +
                        (float) x / width *
                        (
                        mandleBrotDimensions.getxMax() -
                        mandleBrotDimensions.getxMin());
                pixeli
                        = mandleBrotDimensions.getyMin() +
                        (float) y / height *
                        (
                        mandleBrotDimensions.getyMax() -
                        mandleBrotDimensions.getyMin());
```

```
                    Clr = getColor(pixelr, pixeli);
                    if (Clr == -1) {
                        r[x][y] = 255;
                        g[x][y] = 128;
                        b[x][y] = 0;
                    } else {
                        r[x][y] = mandelTables.colorR[Clr %
mandelTables.maxIter];
                        g[x][y] = mandelTables.colorG[Clr %
mandelTables.maxIter];
                        b[x][y] = mandelTables.colorB[Clr %
mandelTables.maxIter];
                    }
                }
            }
```

The output image is shown in Figure 6-1.

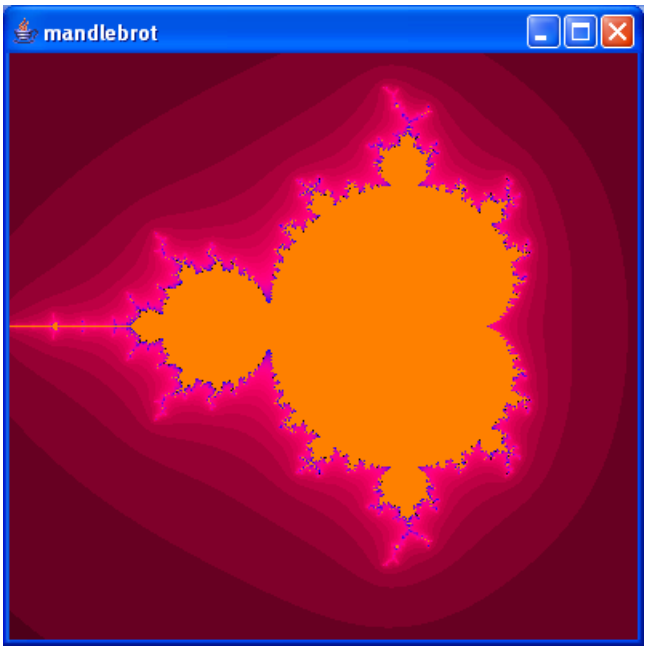


Figure 6-1. Shows the output image from the Von Neumann style program

The next task is to modify the sample program, so the task at hand can be divided into smaller logical parts. The *Mandelbrot* method was overloaded to apply the Mandelbrot set algorithm to a single point as shown in Example 6-3.

Example 6-3

```
public void mandelbrot(int x, int y, short[][] r, short[][] g,
short[][] b) {
        int height = r[0].length;
```

```
            int width = r.length;
            int Clr;
            float pixelr, pixeli;

            pixelr
                    = mandleBrotDimensions.getxMin() +
                    (float) x / width *
                    (
                    mandleBrotDimensions.getxMax() -
                    mandleBrotDimensions.getxMin());
            pixeli
                    = mandleBrotDimensions.getyMin() +
                    (float) y / height *
                    (
                    mandleBrotDimensions.getyMax() -
                    mandleBrotDimensions.getyMin());
            Clr = getColor(pixelr, pixeli);
            if (Clr == -1) {
                r[x][y] = 255;
                g[x][y] = 128;
                b[x][y] = 0;
            } else {
                r[x][y] = mandelTables.•olor[Clr % mandelTables.maxIter];
                g[x][y] = mandelTables.colorG[Clr %
mandelTables.maxIter];
                b[x][y] = mandelTables.colorB[Clr %
mandelTables.maxIter];
            }
        }
```

Using this method we synthesize selected areas of the image as show in Example 6-4 and Figure 6-2.

Example 6-4

```
            short[][] r = sib.getR();
            short[][] g = sib.getG();
            short[][] b = sib.getB();

            int height = 150;
            int width = r.length;

            for (int y = 0; y < height; y++)
                for (int x = 0; x < width; x++) {

                    mandelbrot(x, y, r, g, b);
            }
```

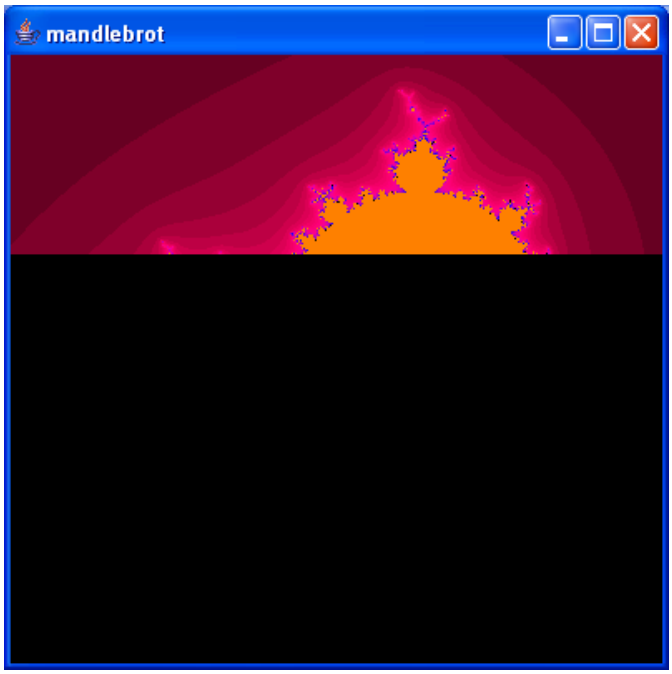


Figure 6-2. Partial processed area of an image

Part of the requirements of the IRJS System is that each single job submitted must generate a jar file with the answer of the job as contents. In this example, it is a partial image. Example 6-5 shows the method used by independent jobs that will work with different sections of the image. Part of the parameters passed to this method is region of the image, and the jar file name for the answer. The result of this method is a jar file containing a section of the image.

Example 6-5

```
public static void computeStrip(Dimension d, Point position, Point
from, Point to, String outputFileName) {

   ShortImageBean sib = new ShortImageBean(d.width, d.height);
   FractalShortImageBean s = new FractalShortImageBean(position,
   from, to);
FractalLogic fl = new FractalLogic();

        s.setR(sib.getR());
        s.setG(sib.getG());
        s.setB(sib.getB());
        System.out.println("Processing piece Image fp=" + from.x + "
tp=" + to.y);
        for (int y = from.y; y < to.y; y++) {
            for (int x = from.x; x < to.x; x++) {
                fl.mandelbrot(x, y, s.getR(), s.getG(), s.getB());
            }
        }
```

```
String ds = SystemUtils.getUserHome() +
SystemUtils.getDirectorySeparator() +
                "rjs" + SystemUtils.getDirectorySeparator();

        File f = new File(ds);
        if (!f.exists()) f.mkdir();
        String fn = ds + outputFileName;
        f = new File(fn);

        Image i = s.getProcessedImage();
        ImageUtils.saveAsPPMJar(i, f);   // saving the image as jar
        System.exit(0);
    }
```

The last step is to create the jobs. For this example we have created eight jobs. Example 6-6 shows the code for one of them. The original program was also modified to read the images from jar files and to build them together to form the original output image.

Example 6-6

```
public class FractalsJob_1 {
    public static void main(String[] args) {
        Point from = new Point(0, 0);
        Point to = new Point(400, from.y + 100);
        Utils.computeStrip(from, to, "Fractals_out1.ppm.jar");
    }}
```

7 SUBMITTING JOBS TO THE IRJS SYSTEM

In this section we describe an example on how to write jobs that can be submitted to the IRJS System. We use the [Mandelbrot] set to create jobs that are suitable for submitting to the IRJS system. The IRJS system accepts tasks or jobs that have the following characteristics:

- They are written in the Java language.
- The class to execute has a *main(...)* method.
- They are independent tasks that do not require any user input during their execution.
- They are known to be large CPU-time consumers.
- They do not require any GUI.
- They are deployed using Java Web Start.
- The outputs of the jobs are written into a jar file.

We have created eight different jobs that use the Mandelbrot algorithms to process a large image. Each job processes a section of the image which horizontal and vertical locations are specified as parameters of the job as shown in Example 7-1.

Example 7-1

```
public class FractalsJob_1 {

    public static void main(String[] args) {
        Point from = new Point(0, 0);
        Point to = new Point(400, from.y + 100);
        Utils.computeStrip(from, to, "Fractals_out1.ppm.jar");
    }
}
```

The jobs are packaged and deployed as Java Web Start applications into a web server using the [Initium] utility. Initium performs a dependency study of the class that will be deployed, in our case class *FractalsJob_1*. It continues to package only the dependencies to this class and generates a jar file. The jar file is signed with the credentials of the IRJS system, and a JNLP file is generated. These two outputs, the signed jar file and the JNLP file, are deployed to the web server where they will be available to the IRJS system. Initium provides an interface that is used to enter all the parameters necessary to package and deploy the jobs as shown in Figure 7-1.

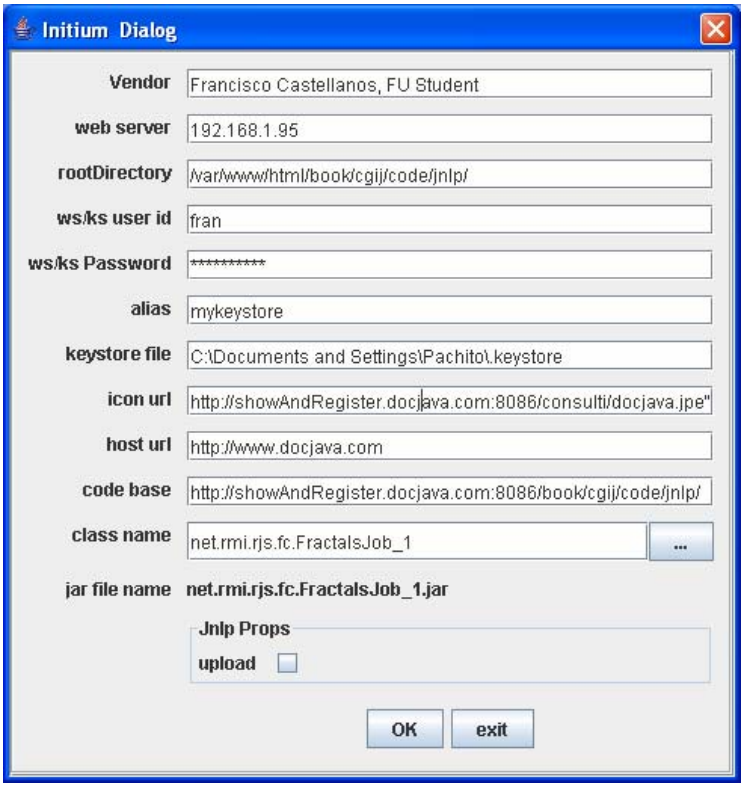


Figure 7-1 Initium Graphical User Interface

The results from the Initium utility are eight Java Web Start applications as shown in figure 7-2. Each JWS application is ready to process a section of an image applying the Mandelbrot algorithms.

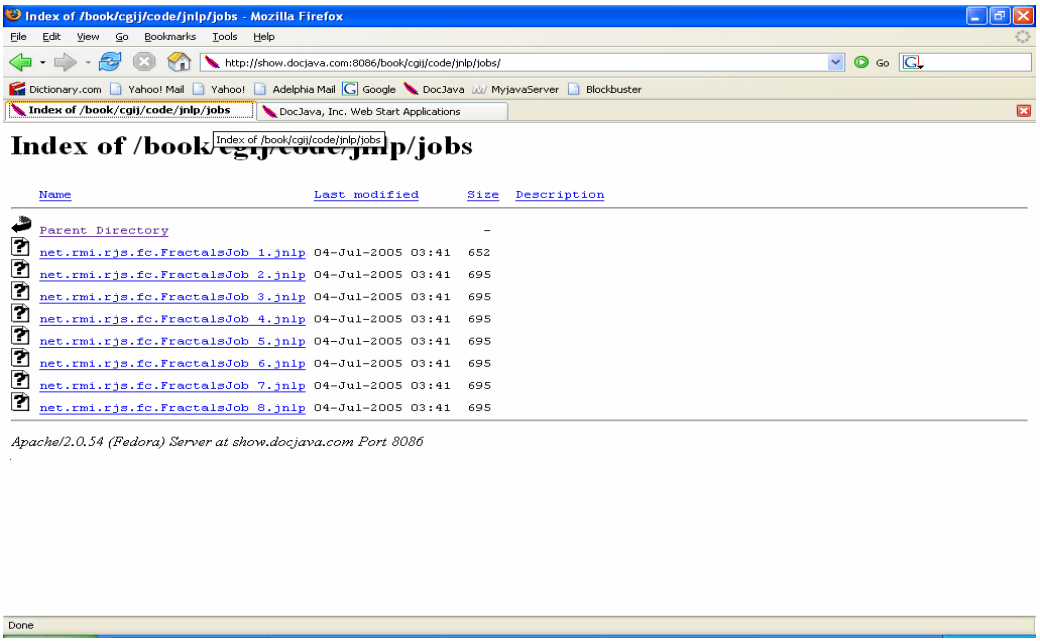


Figure 7-2 Results of Initium: List of eight Java Web Start applications

8 IRJS SYSTEM PROCESSING JOBS

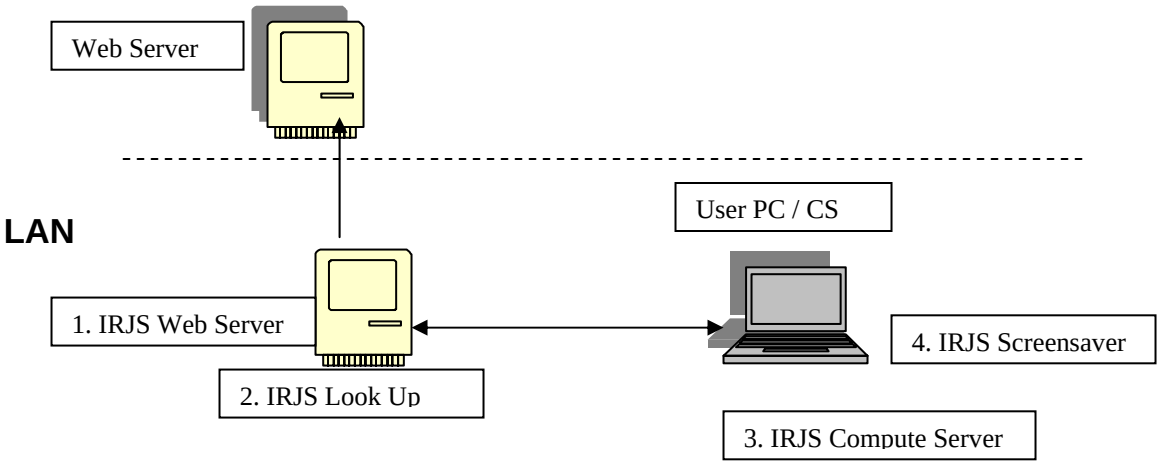


Figure 8-1 IRJS components

The IRJS System is composed of four main modules as depicted in figure 8-1. These four components are stand-alone JWS applications. The first module, the IRJS Web Server

located at [IRJS Web Server], has the responsibility of pulling jobs on demand from an external location (for our example, it is the location where the eight jobs are placed). The second module, the Lookup server (LUS) located at [IRJS LUS], is responsible of managing several activities including job-resource matching, available resources, leasing, and feeding answers back to the web server module. The third component is the Compute Server (CS), located at [IRJS CS]. The CS is triggered by the fourth component, the IRJS screensaver, located at [IRJS SS]. The CS announces its availability and benchmark data to the LUS and waits for jobs to be submitted. Figure 8-2 shows the events mentioned between the IRJS system components.

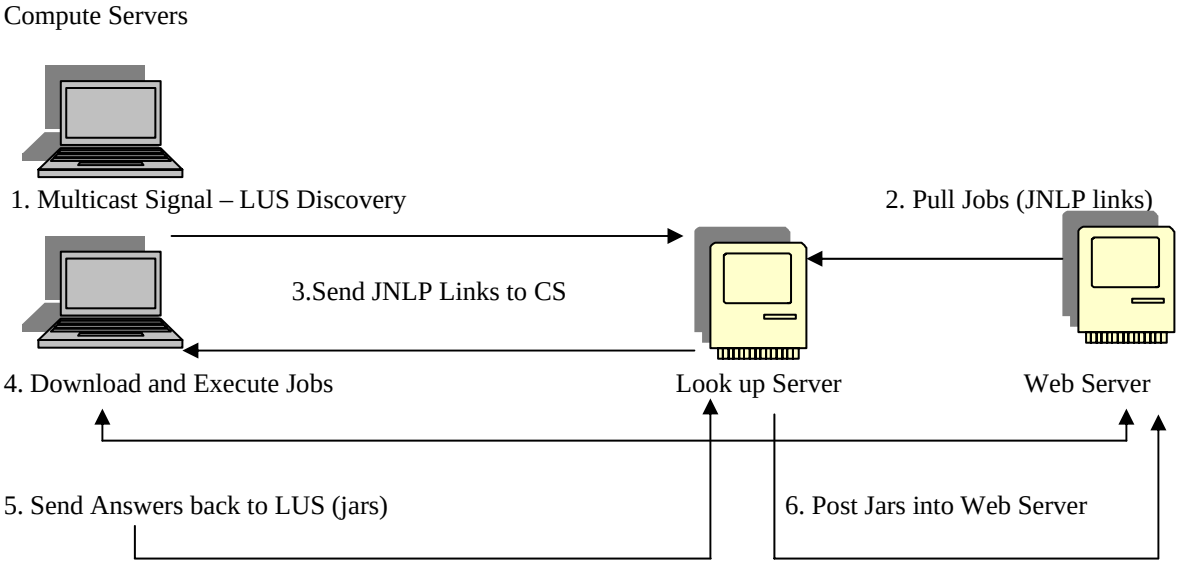


Figure 8-2. IRJS System Events.

For the example at hand, all the components must be launched. We use a separate server to launch the IRJS Web server and the LUS. On separate computers we install the IRJS screensaver. The screensaver display status messages as shown in figures 4-1 and 4-2.

The interface of the LUS is used to start the execution of the system when all components are running. Figure 8-3 shows the LUS interface depicting the available compute servers and their characteristics.

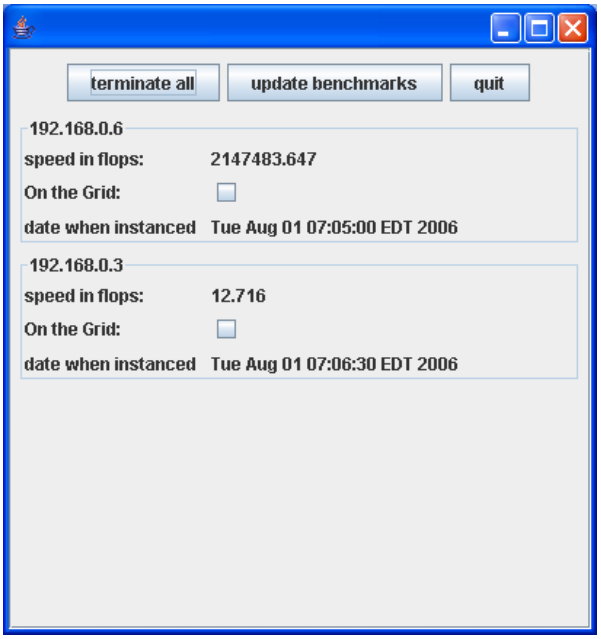


Figure 8-3. LUS interface shows available Compute Servers.

We use the LUS interface to pull available jobs by clicking the *Get Jobs* button, as shown in Figure 8-4. Once the jobs are downloaded they are matched to available CS(s) and deployed. The CS(s) execute the jobs and returns jar files as answers. When the LUS receive the answers, it logs them on the interface, as shown in Figure 8-4.

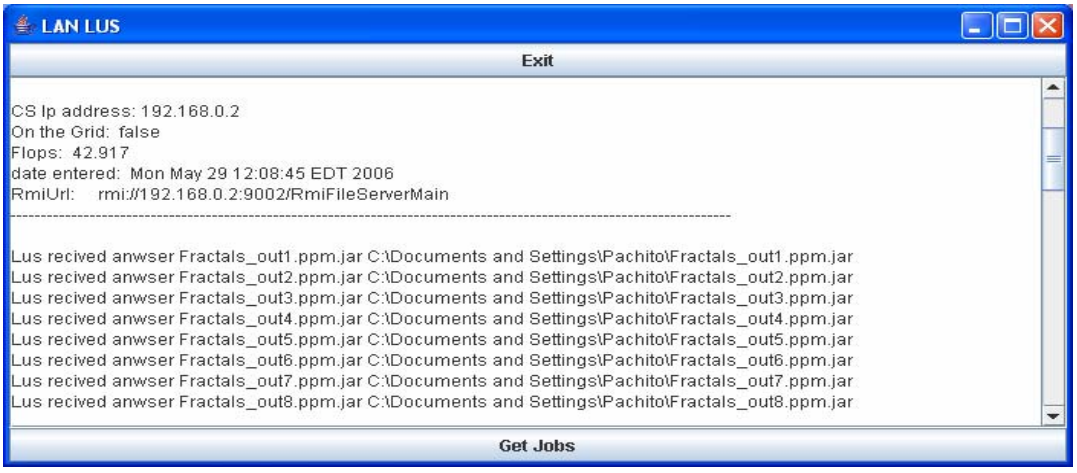


Figure 8-4. Once jobs are completed and answers received, they are logged into the LUS interface.

Lastly, we use the eight answers to build the whole image, as shown in Figures 8-5 and 8-6.

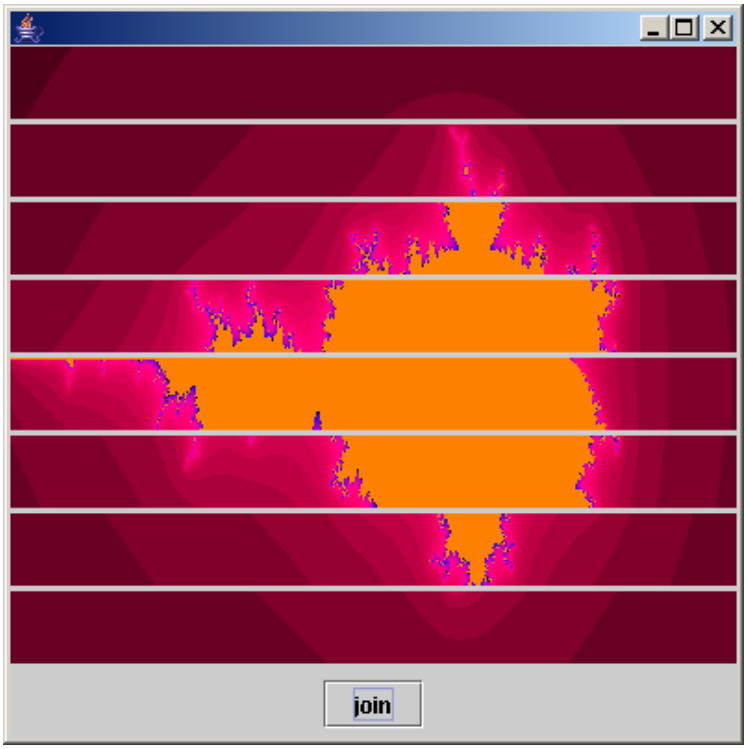


Figure 8-5. This is the set of eight different Images already processed (Mandelbrot Set) by the IRJS System.

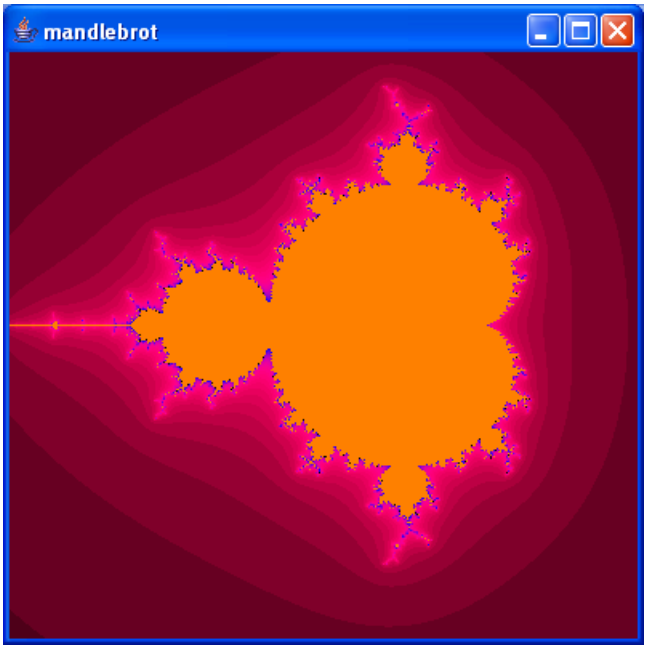


Figure 8-6. The complete processed image.

9 EXPERIMENTAL RESULTS

We experimented the IRJS system to observe the benefits and disadvantages that the system offers. The eight jobs created for the Mandelbrot set example, mentioned in chapter 9, were used to benchmark the IRJS system. Four computers were used, and the IRJS screensaver was installed in each of them for the experiment. To notice the processing time gained, the amount of total processing time for the eight jobs was taken in different scenarios. We first used a single computer and then incremented the amount of computers by one for each experiment. The four computers have the following characteristics:

In order of addition:

A. Linux Fedora Core 4, PIII, 500 MHz, 512 m Ram.
B. Windows XP, Celeron M, 1.5 GHz, 512m Ram
C. Linux Fedora Core 2, Celeron, 500 MHz, 512 Ram
D. Windows XP, P4, 3.0GHz, 1G Ram

Experiment #	Experiment Desc.	Computers	Number of Jobs	Total Processing Time
1	One CS	A	8	2m 53 sec
2	Two CS(s)	A, B	8	1m 7 sec
3	Three CS(s)	A, B, C	8	1 m 2 sec
4	Four CS(s)	A, B, C, D	8	50 sec

Table 9-1. Experimental Results

As shown in Table 9-1, we observed that the amount of processing time decreases as the amount of computers/CS(s) increases. This is perhaps the greatest benefit of the computing grid. Between experiment 1 and 2, the jobs were processed in 1 minute and 46 seconds less, or in ~ 38% of the initial processing time. Between experiment 2 and 3, the jobs were processed in 5 seconds less, or 92% of the processing time in experiment 2. Between experiment 3 and 4 the jobs were processed in 12 seconds less, or ~ 80% of the processing time in experiment 3.

Our second observation is that the IRJS System is as strong as the weakest link. The smallest gain in time was observed after computer C was added to experiment 3. This computer was the slowest computer of the group. In experiment 4, the CS in computer C was the last to complete, 10 seconds later than the rest of the computers, that represents ~ 25% more time, as shown in Figure 9-1.

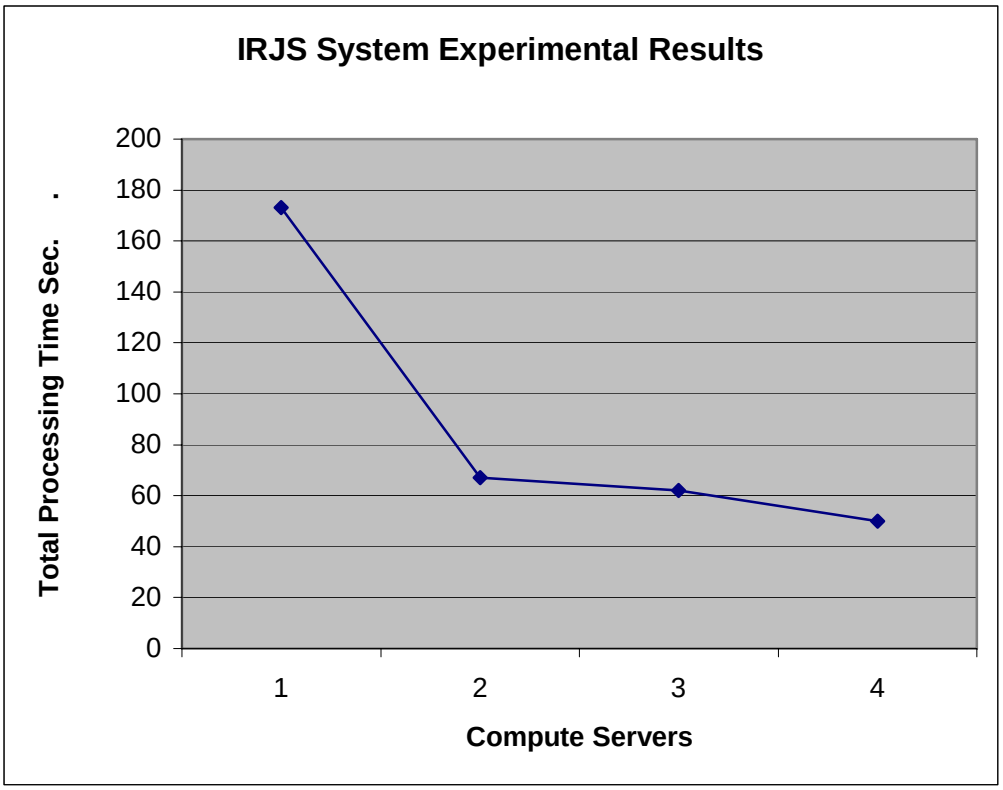


Figure 9-1 Graph of experimental results

10 SUMMARY

A screensaver is capable of detecting user input. We have used this feature to serve as a launching facility for the CS. Through this process the resources of a computer maybe volunteered and utilized close to 100%. The IRJS Screensaver is minimally intrusive, detects user input, and terminates any additional execution of the CS. In addition, we have put to use technologies that promise cross platform solutions such as Java, Java Web Start, and Saverbeans.

REFERENCES

[Lyon and Castellanos] The Initium RJS Screensaver: Part1, MS Windows by Douglas A. Lyon and Francisco Castellanos, in Journal of Object Technology, vol. 5, no. 4, May-June 2006, pp. 7-16.

[Initium] Project Initium: Programmatic Deployment by Douglas A. Lyon, Journal of Object Technology, vol. 3, no. 8, September-October 2004, pp. 55-69

[IRJS CS] http://show.docjava.com:8086/book/cgij/code/jnlp/net.rmi.rjs.pk.main.CSMainForSS.jnl Last accessed September 12, 2006

[IRJS LUS] http://show.docjava.com:8086/book/cgij/code/jnlp/net.rmi.rjs.pk.main.Cr320Lus.jnlp Last accessed September 12, 2006

[IRJS SS] http://show.docjava.com:8086/book/cgij/code/jnlp/net.rmi.rjs.fc.ssinstaller.Main.jnlp Last accessed September 12, 2006

[IRJS Web Server] http://show.docjava.com:8086/book/cgij/code/jnlp/net.rmi.rjs.pk.main.WebServerMain.jnlp Last access September 12, 2006

[Mandelbrot] Mandelbrot set. (2006, August 5). In *Wikipedia, The Free Encyclopedia*. Retrieved 18:44, August 5, 2006, from http://en.wikipedia.org/w/index.php?title=Mandelbrot_set&oldid=67754296.

[Pawel and Lyon] Remote Job Submission Security by Pawel Krepstzul and Douglas A. Lyon, in Journal of Object Technology, vol. 5, no. 1, January-February 2006, pp. 13-29 http://www.jot.fm/issues/issue_2006_01/column2

[SaverBeans] https://jdic.dev.java.net/documentation/incubator/screensaver/index.html Last accessed March 14, 2005.

About the authors

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 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.

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. Email: fsophisco@yahoo.com