JavaPOS Deployment Scenarios
White Paper – Draft

Subject: JavaPOS
Scope: Application Deployment Scenarios
Version: 0.1 – 2004 / 02 / 12

Abstract:

This document will detail 2 possible deployment scenarios for the JavaPOS architecture.

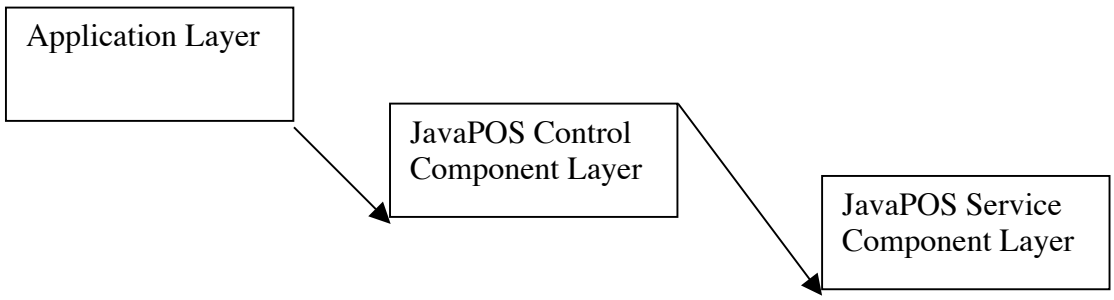
Document History

- Draft version 0.1 updated for JCL 2.2.0 modification
- JCL 2.2.0 released with modifications necessary for Java Web Start deployment
- Java Web Start deployment demonstrated at the June 2003 JavaPOS meeting in Washington DC by Albert Kennis, Star Micronics
- Draft version 0.0 submitted for committee review on 2003 / 09 / 22

JavaPOS Overview

JavaPOS technology is based on the UPOS (Unified Point Of Service) specification, and provides a Java based implementation of that standard suitable for use on Windows, Linux, and other JRE (Java Runtime-time Environment) equipped platforms. JavaPOS provides a mechanism for Java applications to control POS hardware peripherals in a vendor-neutral manner; where a line display from company A and a line display from company B can both be controlled via the same application layer code body. Device vendor neutrality is achieved by utilizing a 3 layer architecture with the lowest layer being provided (in accordance with the UPOS specification) by the hardware device vendors.

JavaPOS based solutions will typically have 3 Java / JavaPOS based architectural layers. These are the application layer, the JavaPOS Control Component layer (implemented in the common jposxx-controls.jar and jcl.jar under the jpos namespace), and the JavaPOS Service Component layer (provided by device manufacturers). Graphically, this architecture looks like this:



Below the JavaPOS Service Component Layer exists the operating system (Windows, Linux, or other) followed by the POS hardware (connected via USB or other). The JavaPOS Service Component layer must encapsulate within itself some mechanism for interfacing with the OS and hardware device. This is often (but not necessarily) done via the JNI (Java Native Interface) mechanism. When JNI is used at the JavaPOS Service Component layer, the provider of this layer will typically provide an operating system specific shared library (eg. windows.dll or linux.so). These shared libraries must be contained in the deployment scenario. The preferred means by which these shared libraries are deployed may vary from one vendor to another. This paper details one possible mechanism for the deployment of these shared libraries.

This document will detail 2 possible deployment scenarios for the JavaPOS architecture. These deployment solutions are detailed from the application layer perspective. Application developers can use these solutions to develop deployment solutions for their JavaPOS based applications.

Deployment Goals

These deployment scenarios are based on the following deployment goals:

- JavaPOS based applications should be able to be packaged as double-clickable applications.
- It should be possible to copy a JavaPOS application contained in a directory or folder and have that application double-clickable usable without the need for installation of other components into the system.
- JavaPOS applications that have been made double-clickable should be easily adapted to work with any available Service Component.
- JavaPOS applications should be capable of being deployed for online and offline usage via the Java Web Start mechanism.

Using these goals as a guide, the following 2 deployment scenarios are proposed.

Deployment Scenario 1 – Double clickable application

This deployment scenario details the steps required to produce a double-clickable JavaPOS application.

The product of this solution will be a directory (folder) containing everything needed to deploy a JavaPOS based application. This folder will contain the JavaPOS application itself, which can be double-clicked to initiate its execution and binding with all JavaPOS architectural layers. The product directory (folder) can be copied onto client computers and then used immediately with no further component installation.

This scenario requires the following:

1. JAR packaged JavaPOS based application
2. JAR packaged JavaPOS Control Component layer – jposxx-controls.jar and jcl.jar
3. JAR packaged JavaPOS Service Components
4. XML registry of JavaPOS devices for processing by the JCL (Java Configuration Loader components) which are incorporated into the JavaPOS Control Component layer

The primary means by which double-clickable JavaPOS applications are created is via the application JAR's manifest file. Within the application JAR's manifest file, the classpath entry is used to bind the application to the other JavaPOS architecture layers.

The process for creating a double-clickable JavaPOS application is:

1. Modify your application's manifest file by adding these keys:
   a. Class-Path: ./jpos17-controls.jar[1] ./jcl.jar[2] ./xercesImpl.jar[3] ./xml-apis.jar ./ ./brandx_printer.jar[4] ./brandy_scanner.jar
   b. Main-Class: com.your-company.YourAppMainClass
2. Create a new directory – "*MyApplicationDeployment*"
3. Copy your application JAR file into this directory
4. Copy jpos17-controls.jar and jcl.jar into this directory

---

[1] jpos17-controls.jar represents the jpos namespace and contains the common device controls and common service interfaces. As of the time of this paper's publication, the current control layer version is 1.7, contained within jpos17-controls.jar. Future revisions will be contained in a simillary named JAR file.
See www.javapos.com

[2] jcl.jar contains the Java Configuration Loader architecture (also implemented under the jpos namespace). This paper requires the use of JCL 2.2.0 or newer.
See www-124.ibm.com/developerworks/projects/jposloader

[3] xercesImpl.jar and xml- apis.jar are the Apache.org XML services implementation that the JCL subsystem requires.
See http://xml.apache.org/xerces2-j/index.html

[4] brandx_printer.jar and brandy_scanner.jar represent the POS hardware vendor provided JavaPOS Service Component layers. The names shown here are only examples; these will vary from hardware vendor to vendor. Add as many of these JAR files are you are planning to use.

5. Copy xercesImpl.jar and xml-apis.jar into this directory
6. Copy any currently available JavaPOS Service Component JAR files into this directory
7. Copy a configured (in accordance with any Service Components from step 4) jpos.xml registry file into this directory.

The keys added to your application jar's manifest file do this: The Class-Path: field tells the JRE that this application requires the jpos17-controls.jar, jcl.jar, xercesImpl.jar, and xml-apis.jar classes be loaded and available for usage. The brandx_printer.jar and brandy_scanner.jar are the service component layer JAR files provided by the device vendors; these too must be available for usage, and so are added to the classpath. The ./ entry is used to add the "MyApplicationDeployment" folder to the classpath. This is done so that the JCL system can find the jpos.xml file contained in this directory. The Main-class entry establishes, in this example, the com.your-company.YourAppMainClass as the JAR's main application, and the one whose main method is executed upon double-clicking.

The "MyApplicationDeployment" directory can now be copied as-is onto any computer or terminal for immediate usage. Once this directory is copied, the application JAR file can be double-clicked and used with no further action required.

Note to Service Component layer providers:

In order for this deployment scenario to be possible, the Service Component layer JARs must encapsulate any required operating specific shared libraries. These libraries must be automatically extracted to the local machine before being bound to.

Deployment Scenario 2 – Java Web Start application

This deployment scenario details the steps required to produce an application that can be distributed and used with the Java Web Start mechanism.

The product of this solution will be a JNLP application description file used by the Java Web Start system to describe the application and its architecture.

This scenario requires the following:

1. JAR packaged JavaPOS based application
2. JAR packaged JavaPOS Control Component layer – jposxx-controls.jar and jcl.jar
3. JAR packaged JavaPOS Service Components
4. JAR packaged operating system specific shared libraries used by the JavaPOS Service Components
5. XML registry of JavaPOS devices for processing by the JCL (Java Configuration Loader components) which are incorporated into the JavaPOS Control Component layer

We need to create the JNLP file, which describes the Java Web Start application. The process of creating this file will now be described. A complete example is listed in Appendix 1 of this document.

The following keys have been excerpted from the example JNLP file. These keys are shown here because they are specific to using the Java Web Start mechanism for a JavaPOS application and of particular importance.

<jnlp spec="1.0+" codebase="http://your.server.com/javapos-jws/apps" href="your-javapos-app.jnlp">

This key specifies that this Java Web Start application is rooted at the server's "http://your.server.com/javapos-jws/apps" directory and that it is being described by the "your-application.jnlp" file.

<jnlp>…<information><offline-allowed/>…

Within the <information> record, the <offline-allowed/> entry specifies that this Java Web Start application should be available for use even when a network connection is not available. If this policy is not suitable for your application, simply eliminate this tag.

<jnlp>…<security><all-permissions/></security>

This entry specifies that this Java Web Start application requires complete access to all system resources. This is required so that the Service Component layers can access the physical POS devices through, for example, the computer's USB bus. Specifying this requires your application JAR and all other JARs involved in the deployment be signed via the JDK's jarsigner application or similar mechanisms. See appendix 2 for the sequence of steps required to "self sign" JAR files, which is a useful technique during development and testing or if you're poor like me.

```
<jnlp>
    …
    <resources>
        <jar href="lib/YourApplication.jar"/>
        <jar href="lib/brandx_printer.jar"/>
        <jar href="lib/brandx_scanner.jar"/>
        <jar href="lib/jpos17-controls.jar"/>
        <jar href="lib/jcl.jar"/>
        <jar href="lib/xercesImpl.jar"/>
        <jar href="lib/xml-apis.jar.jar"/>
        <jar href="lib/javapos-registry.jar"/>
        <j2se version="1.4+"/>
    </resources>
```

The <resources> record specifies all JAR files that are involved in this deployment and their locations relative to the previously defined application deployment root location. In this example, the first JAR specified is the one containing your application. The following two JAR files are those provided by JavaPOS Service Component vendors, and the ones responsible for actually accessing the POS device hardware. The following JAR files, jpos17-controls.jar and jcl.jar, contain the Control Component layer and the JCL subsystem. The following two JAR files, xercesImpl.jar and xml-apis.jar, are the Apache.org Xerces XML services system, used by the JCL to process the jpos.xml file. The final JAR file, javapos-registry.jar, contains your jpos.xml registry file in its root directory. The JCL system looks for the jpos.xml file only in the root directory of JAR files, so be sure to place this file there.

The <j2se …> tag specifies that this deployment requires a version of the j2se system concurrent with or newer then version 1.4. This would, for example, allow this application to be run under either a 1.4.1,a 1.4.x, or a 1.5.x j2se product.

```
<jnlp>
    …
    <resources os="Windows">
        <nativelib href="lib/brandx_printer_windows_libs.jar"/>
        <nativelib href="lib/brandy_scanner_windows_libs.jar"/>
      </resources>
    <resources os="Linux">
        <nativelib href="lib/brandx_printer_linux_libs.jar"/>
        <nativelib href="lib/brandy_scanner_linux_libs.jar"/>
      </resources>
```

When using Java Web Start, components that use operating system specific shared libraries via JNI must be shipped with operating specific JAR files containing those shared libraries. These JAR files should contain, in their root directories, only those shared libraries used on a single operating system. Thusly, there will be one JAR file per operating system. In this example, the shared libraries used by the components of brandx_printer.jar are placed into brandx_printer_windows_libs.jar and brandx_printer_linux_libs.jar. All JAR files for use on Windows are listed under the os=”Windows” specific resource record and those for use on Linux are listed

similarly.

These tags are the ones specific to deploying a JavaPOS application via Java Web Start. Other tags are also required, but these are not specific to JavaPOS, and so are not described in detail here. A complete JNLP file containing all tags is listed in Appendix 1 of this document.

Save this xml structure into the file, "your-javapos-app.jnlp" or a name suitable for you.

Java Web Start applications are launched when a user clicks on a hyperlink pointing to a JNLP file. A suitable hyperlink for this example is:

```
<font size="20"><a href="apps/your-javapos-app.jnlp">Launch
Application</a></font>
```

Prior to deploying your Java Web Start application, you must digitally sign all JAR files. This can be done via the Java SDK included jarsigner application or another similar application. Appendix 2 of this document gives the instructions for self-signing JAR files. In an actual deployment though you would want to obtain a commercial signature from Verasign, the Thawte Group, or another such entity.

You must now create the following directory / file system structure on your server:

http://your.server.com/javapos-jws
- launch-page.html
- /apps
  - your-javapos-app.jnlp
  - /lib
    - YourApplication.jar
    - Brandx_printer.jar
    - Brandy_scanner.jar
    - jpos17-controls.jar
    - jcl.jar
    - xercesImpl.jar
    - xml-apis.jar
    - javapos-registry.jar
    - brandx_printer_windows_libs.jar
    - brandx_printer_linux_libs.jar

Copy this directory and file system structure over to your server and verify that your JavaPOS application is now available for use via the Java Web Start mechanism.

Appendix 1: Example JNLP file for JavaPOS application deployment

This JNLP follows from the Java Web Start example described previously.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for JavaPOS application -->
<jnlp spec="1.0+" codebase="http://your.server.com/javapos-jws/apps" href="your-
      javapos-app.jnlp">
  <information>
    <title>Your JavaPOS Application via Java Web Start</title>
    <vendor>Your Company</vendor>
    <homepage href="docs/help.html"/>
    <description>JavaPOS Deployment via Java Web Start</description>
    <description kind="short">Complete application - JavaPOS - JCL - Service
     Object Layer deployment.</description>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <jar href="lib/YourApplication.jar"/>
    <jar href="lib/brandx_printer.jar"/>
    <jar href="lib/brandx_scanner.jar"/>
    <jar href="lib/jpos17-controls.jar"/>
    <jar href="lib/jcl.jar"/>
    <jar href="lib/xercesImpl.jar"/>
    <jar href="lib/xml-apis.jar"/>
    <jar href="lib/javapos-registry.jar"/>
    <j2se version="1.4"/>
  </resources>
  <resources os="Windows">
    <nativelib href="lib/brandx_printer_windows_libs.jar"/>
    <nativelib href="lib/brandy_scanner_windows_libs.jar"/>
  </resources>
  <resources os="Linux">
    <nativelib href="lib/brandx_printer_linux_libs.jar"/>
    <nativelib href="lib/brandy_scanner_linux_libs.jar"/>
  </resources>
  <application-desc>
  </application-desc>
</jnlp>
```

Appendix 2: Jar signing / self-signing process steps

The following sequence of steps can be used to sign your deployments Jar files (as required for JavaPOS application deployment via the Java Web Start mechanism). This causes your application's Jar files to be "self-signed." Although this is free and good for testing, it is best to obtain digital signatures for use in the signing process from entities such as VeraSign or others.

1. Make sure that you have the JDK's **keytoo**l and **jarsigner** applications in your path. These tools are located in the SDK bin directory.

2. Create a new key in a new keystore as follows:

   keytool -genkey -keystore myKeystore -alias myself

   You will get prompted for a information about the new key, such as password, name, etc. This will create the myKeystore file on disk.

3. Then create a self-signed test certificate as follows:

   keytool -selfcert -alias myself -keystore myKeystore

   This will prompt for the password. Generating the certificate may take a few minutes.

4. Finally, sign the JAR file with the test certificate as follows:
   jarsigner -keystore myKeystore test.jar myself

Document End