JAVA™ NETWORK LAUNCHING PROTOCOL & API SPECIFICATION (JSR-56)

VERSION 1.0.1

Java Software
A Division of Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, California 94303
415 960-1300 fax 415 969-9131

May 21, 2001

René W. Schmidt

Java(TM) Network Launching Protocol (JNLP) Specification ("Specification")
Version: 1.0.1
Status: FCS
Release: May 21, 2001

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control and General Terms as set forth in Sun's website Legal Terms. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's intellectual property rights that are essential to practice the Specification, to internally practice the Specification solely for the purpose of creating a clean room implementation of the Specification that: (i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification, as defined by Sun, without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by Sun in the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages (or other packages defined by Sun); (vi) satisfies all testing requirements available from Sun relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any Sun source code or binary code materials; and (viii) does not include any Sun source code or binary code materials without an appropriate and separate license from Sun. The Specification contains the proprietary information of Sun and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, and the Java Coffee Cup Logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#86972/Form ID#011801)

TABLE OF CONTENTS

0 Preface

This document, the Java™ Network Launching Protocol and API Specification, v1.0.1, is also known as the JNLP Specification. In addition to this specification, the Java Network Launching API has Javadoc documentation (referred to as the JNLP API Reference, v1.0) and a reference implementation for public download at the following location:

```
http://java.sun.com/products/javawebstart/
```

The reference implementation provides a behavioral benchmark. In the case of a discrepancy, the order of resolution is this specification, then the JNLP API Reference, v1.0, and finally the reference implementation.

0.1 Who Should Read This Specification

This document is intended for consumption by:

- Software vendors that want to provide an application or utility that conforms with this specification.
- Web Authoring Tool developers and Application Tool developers that want to provide tool support that conforms to this specification.
- Sophisticated Web authors and Web site administrators who want to understand the underlying mechanisms of the Java Network Launching technology.

Please note that this specification is not a User's Guide and is not intended to be used as such.

0.2 API Reference

The JNLP API Reference, v1.0, provides the complete description of all the interfaces, classes, exceptions, and methods that compose the JNLP API. Simplified method signatures are provided throughout this specification. Please refer to the API Reference for the complete method signatures.

0.3 Other Java Specifications

The following Java API Specifications are referenced throughout this specification:

- Java 2 Platform Standard Edition, v1.2 and v1.3 (J2SE). The specifications can be found at:

  ```
  http://java.sun.com/j2se/
  ```

- Java 2 Platform Enterprise Edition, v1.2 (J2EE). The specification can be found at:

  ```
  http://java.sun.com/j2ee/
  ```

0.4 Other Important References

The following Internet Specifications provide relevant information to the development and implementation of the JNLP Specification and tools that support the specification.

- RFC 1630 Uniform Resource Identifiers (URI)
- RFC 1738 Uniform Resource Locators (URL)
- RFC 1808 Relative Uniform Resource Locators
- RFC 1945 Hypertext Transfer Protocol (HTTP/1.0)
- RFC 2045 MIME Part One: Format of Internet Message Bodies
- RFC 2046 MIME Part Two: Media Types
- RFC 2047 MIME Part Three: Message Header Extensions for non-ASCII text
- RFC 2048 MIME Part Four: Registration Procedures
- RFC 2049 MIME Part Five: Conformance Criteria and Examples
- RFC 2616 Hypertext Transfer Protocol (HTTP/1.1)

You can locate the online versions of any of these RFCs at:

`http://www.rfc-editor.org/`

The World Wide Web Consortium (`http://www.w3c.org`) is a definitive source of HTTP related information that affects this specification and its implementations.

The Extensible Markup Language (XML) is utilized by the JNLP Descriptor described in this specification. More information about XML can be found at the following websites:

`http://www.w3.org/`

`http://www.xml.org/`

0.5 Providing Feedback

The success of the Java Community Process depends on your participation in the community. We welcome any and all feedback about this specification. Please e-mail your comments to:

jnlp-comments@eng.sun.com

Please note that due to the volume of feedback that we receive, you will not normally receive a reply. However, each and every comment is read, evaluated, and archived by the specification team.

0.6 Acknowledgments

The success of the Java Platform depends on the process used to define and refine it. This open process permits the development of high quality specifications in internet time and involves many individuals and corporations.

Many people have contributed to this specification and the reference implementation. Thanks to:

- The following people at Sun Microsystems: Georges Saab, Lars Bak, Tim Lindholm, Tom Ball, Phil Milne, Brian Beck, Norbert Lindenberg, and Stanley Man-Kit Ho.

- The members of the JCP expert group (in particular Alex Rosen of SilverStream Software) and participants who reviewed this document.

- The people on the Internet that reviewed the first public draft of this specification.

A special thanks to my friends in the JNLP team at the Java Software Division at Sun Microsystems, Steve Bohne, Andrey Chernyshev, Andy Herrick, Hans Muller, Kumar Srinivasan, Scott Violet, and Nathan Wang, who did most of the hard work to get this project started, shaped, and delivered.

0.7 Revision History

0.7.1 Changes since release 1.0

This is a minor update of the version 1.0 specification. This specification update contains no changes nor additions to the JNLP file or the JNLP API. This update addresses several inconsistencies and typos in the original specification, as well as one Applet compatibility issue. The major changes are described below:

- Update the untrusted environment to include the AWT permission *accessEventQueue*. This is to comply with the Applet sandbox model.

- Clarified the use of encoded/unencoded URLs in a JNLP file.

- Clarified that the JARDiff index file uses the *remove* command and not the *delete* command

- Fixed minor typos and inconsistencies in the examples.

This revision does not introduce new version numbers for the JNLP file nor the JNLP API. A JNLP Client implementing this specification must be able to run a JNLP file which requires 1.0, i.e., the spec attribute in the jnlp element is set to 1.0.

1 OVERVIEW

The Java Network Launching Protocol and API (JNLP) is a Web-centric provisioning[1] protocol and application environment for Web-deployed Java 2 Technology-based applications. An application implementing this specification is called a JNLP Client.

The main concepts in this specification are:

- A Web-centric application model with no installation phase, which provides transparent and incremental updates, as well as incremental downloading of an application. This is similar to the model for HTML pages and Applets, but with greater control and flexibility.

- A provisioning protocol that describes how to package an application on a Web server, so it can be delivered across the Web to a set of JNLP Clients. The key component in this provisioning protocol is the JNLP file, which describes how to download and launch an application.

- A standard execution environment for the application. The execution environment includes both a safe environment where access to the local disk and the network is restricted for untrusted applications, and an unrestricted environment for trusted applications. The restricted environment is similar to the well-known Applet sandbox, but extended with additional APIs.

The main concepts are introduced in the following sections.

1.1 WEB-CENTRIC APPLICATION MODEL

A JNLP Client is an application or service that can launch applications on a client system from resources hosted across the network. It is not a general installation protocol for software components. A high-level view of a JNLP Client is that it allows an application to be run from a codebase that is accessed over the Web, rather than from the local file system. It provides a facility similar to what would happen if URLs were allowed in the JRE's classpath, e.g., something that looks like this:

```
java -classpath http://www.mysite.com/app/MyApp.jar com.mysite.app.Main
```

The above example illustrates the basic functionality of a JNLP Client. JNLP goes further than this, however. First, it provides the ability to specify which version of the Java 2 Platform (JRE) that the application requires. In the above example, this amounts to choosing what `java` command to use. If the requested JRE version is not available, a JRE can be downloaded and installed automatically[2]. Second, it provides the ability to specify native libraries as part of the application. Native libraries are downloaded in JAR files. Thus, both signing and compression of the libraries are supported. The native libraries are loaded into the running process using the `System.loadLibrary` method.

All the resources that an JNLP Client needs to access in order to launch an application are referenced with URLs. Conceptually, all of the application's resources reside on the Web server. A JNLP Client is allowed and encouraged to cache resources that are downloaded from the Web. This will improve consecutive startup times, minimize network traffic, and enable offline operation.

This application model provides the following benefits:

---

1 The term *provisioning* is commonly used to denote the distribution of software components, such as an application, from a central server to a set of client machines. This is sometime also referred to as deployment of an application.

2 The provisioning protocol defined in this specification also allows a JRE to be packaged on a Web server for automatic installation on the client machine by a JNLP Client.

- **No installation phase:** A JNLP Client simply needs to download and cache the application's resources. The user does not need to be prompted about install directories and the like.

- **Transparent update:** A JNLP Client can check the currently cached resources against the versions hosted on the Web Server and transparently download newer versions.

- **Incremental update:** The JNLP Client only needs to download the resources that have been changed when an application is updated. If only a few of the application's resources have been modified, this can significantly reduce the amount of data that needs to be downloaded when upgrading to a new version of an application. Furthermore, incremental update of individual JAR files is also supported.

- **Incremental download:** A JNLP Client does not need to download an entire application before it is launched. For example, for a spreadsheet application the downloading of the graphing module could be postponed until first use. JNLP supports this model by allowing the developer to specify what resources are needed before an application is launched (*eager*), and what resources can be downloaded later (*lazy*). Furthermore, JNLP provides an API so the developer can check if a resource is local or not (e.g., need to be downloaded or not), and to request non-local resources to be downloaded.

- **Offline support:** A JNLP Client can launch an application offline if a sufficient set of resources are cached locally. However, most applications deployed using JNLP are expected to be Web-centric, i.e., they will typically connect back to a Web server or database to retrieve their state. Hence, many applications will only work online. The application developer specifies if offline operation is supported, and what resources are needed locally to launch the application offline.

1.2 PROVISIONING

1.2.1 JNLP FILE

The core of the JNLP technology is the JNLP file. The JNLP file is an XML document.

Most commonly, a JNLP file will describe an application. A JNLP file of this kind is called an *application descriptor*. It specifies the JAR files the application consists of, the Java 2 platform it requires, optional packages that it depends on, its name and other display information, its runtime parameters and system properties, etc. There is a one-to-one correspondence between an application descriptor and an application.

A JNLP file does not contain any binary data itself. Instead it contains URLs that point to all binary data, such as icons (in JPEG or GIF format), and binary code resources, such as Java classes and native libraries (contained in JAR files). Figure 1 illustrates how an application is described with JNLP files. The root JNLP file (application descriptor) contains the basic information such as name and vendor, main class, and so forth. The JAR files that constitute the "classpath" for the application are all referred to with URLs.

A JNLP file can also refer to other JNLP files, called *extension descriptors.* An extension descriptor typically describes a component that must be used in order to run the application. The resources described in the extension descriptor become part of the classpath for the application. This allows common functionality to be factored out and described once. An extension descriptor also provides the ability to run an installer that can install platform-dependent resources before the application is launched, e.g., to install device drivers.

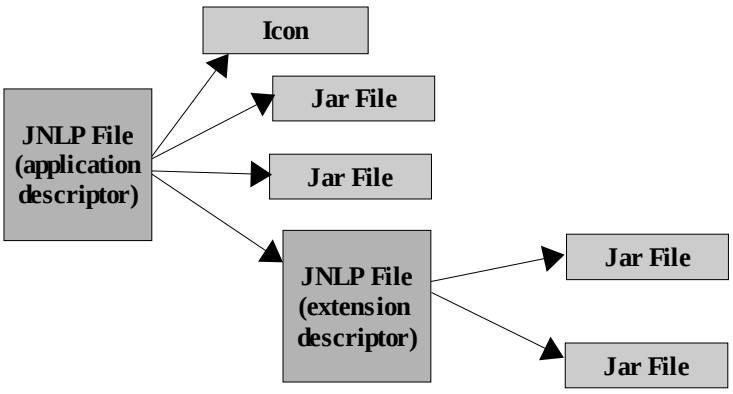

Figure 1: JNLP File and External Resources

The JNLP file is, in some sense, similar to a traditional executable format. Traditionally, applications are delivered as binary platform-dependent files. For example, on Windows, an application is delivered as a `MyApp.exe` executable. The executable format is designed so the Windows operating system can load the application and execute it. It also contains information about external dependencies, such as, e.g., `MyApp.dll`. This format is file-centric; all external references are references to files on the local file system. In contrast, a JNLP file does not contain any binary data itself, but instead contains URLs to where they can be obtained from. The JNLP file format is Web-centric; the references to external resources are URLs, instead of file names.

1.2.2 Downloading Resources

The JNLP Client can download 3 different kind of resources: JAR files, images, and JNLP files. All resources in a JNLP file are uniquely named using either a URL or a URL/version-id pair. A typical application deployed using JNLP will consist of a set of JAR files and a set of images[3]. JAR files, images, and JNLP files can be downloaded using standard HTTP GET requests. For example:

```
http://www.mysite.com/app/MyApp.jar
```

This basic download protocol works from a standard unmodified Web server. This leverages existing Web server technology, which is important to achieve wide-spread use of a new technology on the Internet.

To provide more control and better utilization of bandwidth, a version-based download protocol is also supported. The version-based protocol is designed to:

- Allow several versions of an application to co-exist on a server at a given time. In particular, this means that an application that is distributed as several JAR files can be safely upgraded. A JNLP Client that is downloading JAR files right when a Web server is being updated will never download JAR files that are a mix between two application versions.

- Provide a unique URL for an application independent of its version. This allows a JNLP Client to automatically detect and flush old versions out of the cache.

- Make it possible to incrementally update already-downloaded JAR files. This can substantially minimize the download requirements for upgrading to a new version.

3 The image files described in the JNLP file are icons that can be used by the JNLP Client to integrate the application into the desktop environment. They are not for use by the application itself. All application resources, such as images, must generally either be included in one of the JAR files or be explicitly downloaded using, e.g, an HTTP request.

- Allow users to stick with a given version rather than always getting the latest version from the Web server. For example, a JNLP Client can download an updated version in the background, while the already-downloaded version is being used.

The version-based protocol requires special support on the Web server. This support can be provided using servlets, CGI-scripts, or by similar means.

The use of the version-based protocol is specified in the JNLP file on a per-resource basis. Depending on the facilities the Web server offers (and possibly other factors), the application developer can choose whether the version-based protocol should be used or not.

1.3 Application Environment

The application environment defines a common set of services and system settings that an application launched with a JNLP Client can depend on. The core of this environment is the Java 2 Platform Standard Edition. In addition, this specification defines additional APIs and settings:

- Configured HTTP proxies.
- A secure execution environment that is similar to the well-known Applet sandbox.
- An API to securely and dynamically lookup and access features on the client platform, such as instructing the default browser to display a URL.

The application environment is defined as a set of required services that must be implemented by all implementations that conform to this specification, and a set of optional services that are not required to be implemented. Applications must check for the presence of optional services and handle their absence sensibly.

1.4 An Example

A helper application that implements the Java Network Launching protocol and API can be associated with a Web browser. The helper application gets configured with the proper HTTP proxy settings during installation, so they can be passed along to a launched application[4]. Thus, the user does not have to specify proxy settings for each application separately.

When a user clicks on a link pointing to a JNLP file, the browser will download the file and invoke the helper application with the name of the downloaded file as an argument. The helper application (i.e., the JNLP Client) interprets the JNLP file, which will direct it to download and locally cache the JAR files and other resources for the particular application. When all required JAR files have been downloaded, the application is launched.

A sample JNLP file, which is an XML document, is shown here:

4 In Sun's Java 2 SE JREs, proxy settings can be specified using the `proxyHost` and `proxyPort` system properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="http://www.mysite.com/app">
  <information>
    <title>Draw!</title>
    <vendor>My Web Company</vendor>
    <icon href="draw-icon.jpg"/>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.3+"/>
    <jar href="draw.jar"/>
  </resources>
  <application-desc main-class="com.mysite.Draw"/>
</jnlp>
```

The JNLP file describes how to launch the sample application, titled *Draw!*. In the JNLP file, it is specified that the Java 2 platform, version 1.3 or higher is required to run this application, along with some general application information that can be displayed to the user during the download phase.

1.5 Comparing JNLP with Other Technologies

The JNLP technology is related to Java Applets. Java Applets are automatically downloaded, cached, and launched by a Web browser without requiring any user interaction, and Applets are executed in a secure sandbox environment by default. Applets are a core part of the Java 2 SE. Many of the technologies that are used by JNLP are borrowed from the Applet technology, such as the downloading of code and the secure sandbox.

Applications launched with JNLP do not run inside a browser window, but are instead separate applications that are run on separate Java Virtual Machines (JVMs). Thus, applications launched with JNLP are typically more like traditional desktop applications that are commonly distributed as shrink-wrapped software, e.g., on CDs.

JNLP is not a general installer for applications. It is particularly targeted to Web-deployed Java Technology-based applications, i.e., applications that can be downloaded from the Web and which store most of their state on the Web.

The JNLP protocol defines how Java Runtime Environments and optional packages can be installed automatically. This will typically require the JREs and optional packages to be bundled in a traditional installer.

2 TERMS USED

Term	Description
JRE	Java 2 Standard Edition Runtime Environment
JVM	Java Virtual Machine
JNLP Client	A software application or service that implements this specification.
Application	The term *application* refers to the Java application or Java Applet that is launched by a JNLP Client.
Extension	The term *extension* denotes a JNLP file that encapsulates a set of code resources, such as a optional package or a JRE itself.
Version-id	A specification of an exact version, e.g., 1.2. See also Appendix A.
Version string	A specification of a key that is used to match the version-id's. For example, "1.2.2* 1.3.0" is a Version string that will match the version-id's 1.2.2-w, 1.2.2.0, 1.3.0, and so forth. See also Appendix A.

3 JNLP FILE

The core of the JNLP technology is the JNLP file. The JNLP file describes how to download and launch a particular application.

The description of the JNLP file is split into functional categories. Thus, each section typically does not describe all subelements or attributes of a given element. To view the complete set of attributes for an element, the set of subelements, and a brief description of each, see Appendix C, which contains the formal syntax of the JNLP file in the form of an annotated XML DTD.

3.1 OVERVIEW

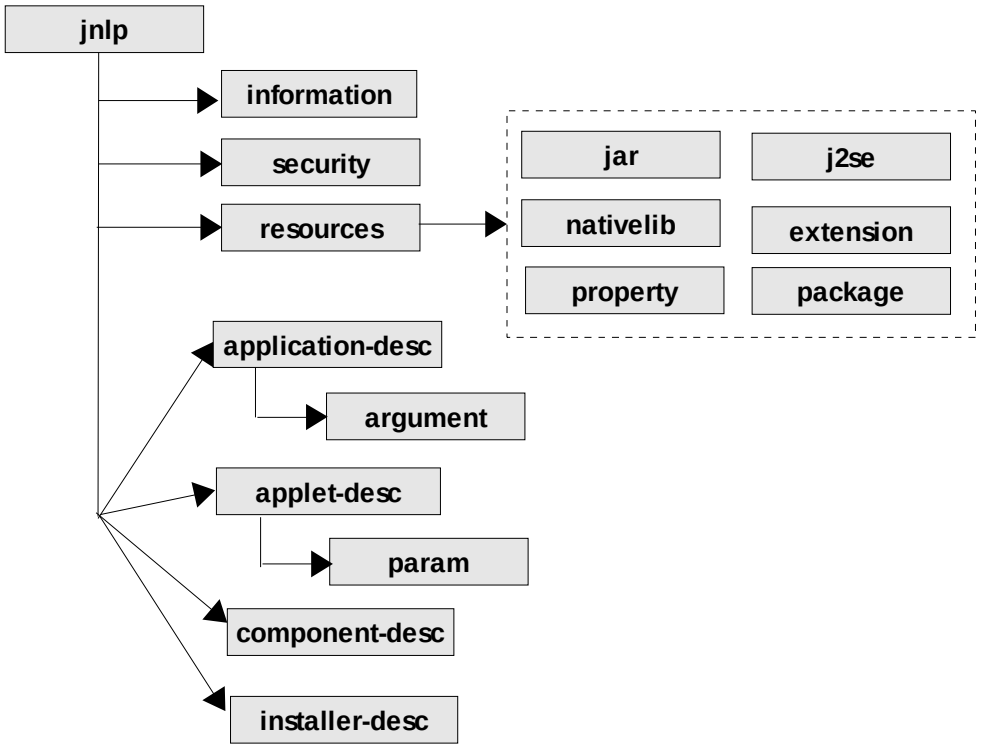

Figure 2: Overview of a JNLP file with the most common elements shown.

Figure 2 shows the outline of a JNLP file. It has 5 main sections:

- The jnlp element is the root element. It has a set of attributes that are used to specify information that is specific to the JNLP file itself.
- The information element describes meta-information about the application. That information can, for example, be shown to the user during download. This is explained later in this section.
- The security element is used to request a trusted application environment. This is described in detail in Section 5.3.

- The resources element specifies all the resources that are part of the application, such as Java class files, native libraries, and system properties. Section 4 describes this in detail.

- The final part of a JNLP file is one of the following four elements: application-desc, applet-desc, component-desc, and installer-desc. Only one of the four can be specified in each JNLP file. A JNLP file with either an application-desc or applet-desc is called an application descriptor, whereas a JNLP file with an component-desc or an installer-desc element is called an extension descriptor. These elements are described later in this section.

The following JNLP file fragment shows the outline with the actual syntax for a JNLP file:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="http://www.mysite.com/application/" ...>
  <information> ... </information>
  <security> ... </security>
  <resources> ... </resources>
  <application-desc> ... </application-desc>
</jnlp>
```

The jnlp element contains the spec attribute that specifies the versions of the specification that this JNLP file requires. The value of the attribute is specified as a version string. If none of the versions of the specification that the JNLP Client implements matches the version string, then the launch should be aborted. If the attribute is not explicitly defined, it must be assumed to be "1.0+", i.e., the JNLP file works with a JNLP Client that supports the 1.0 specification and higher (i.e., it works with all JNLP Clients. See Appendix A).

3.2 MIME Type and Default File Extension

The default MIME type and extension that should be associated with a JNLP file are shown in the following table:

Default MIME Type	Default Extension
application/x-java-jnlp-file	.jnlp

3.3 Parsing a JNLP Description

It is expected that future versions of this specification will introduce new elements and attributes that would be backwards-compatible with the current DTD. Thus, a JNLP Client should not reject a JNLP file that has extra attributes or elements. This means that the JNLP Client's XML parser must not validate the JNLP XML file against any fixed version of the JNLP DTD. However, like any XML parser, if the JNLP XML file contains a DOCTYPE declaration that specifies which DTD it uses, the parser may choose to validate the JNLP file against that specified DTD. If the JNLP file does not contain a DOCTYPE declaration, the parser may not validate the file against any DTD.

3.4 References to external resources

All references to external resources in a JNLP file are specified as URLs using the href attribute. For example:

```
<icon href="http://www.mysite.com/images/icon.gif">
```

```
<jar href="classes/MyApp.jar">

<jnlp href="http://www.mysite.com/App.jnlp">
```

An href element can either contain a relative URL or an absolute URL as shown above. A relative URL is relative to the URL given in the `codebase` attribute of the `jnlp` root element. For example:

```
<jnlp codebase="http://www.mysite.com/application/" ... >
```

A relative URL cannot contain parent directory notations, such as "..". It must denote a file that is stored in a subdirectory of the codebase. URLs in a JNLP file should always be properly encoded (also known as "escaped" form in RFC 2396 Section 2.4.2), e.g., a space should be represented as %20 in a HTTP URL. A JNLP Client must used the URL exactly as specified in the JNLP file when making a request to the Web server (See also Section 6.1).

All resources can also be specified using a URL and version string pair. Thus, all elements that support the href attribute also support the version attribute, which specifies the version of the given resource that is required. For example,

```
<jar href="classes/MyApp.jar" version="1.2">
```

The version attribute can not only specify an exact version, as shown above, but can also specify a list of versions, called a Version string. Individual version-id's are separated by spaces. The individual version-id's in a Version string can, optionally, be followed by either a star (*) or a plus sign (+). The star means prefix match, and the plus sign means this version or greater. For example:

```
<jar href="classes/MyApp.jar" version="1.3.0 1.2.2*">
```

The meaning of the above is: the JAR file at the given URL that either has the version-id 1.3.0 or has a version-id where 1.2.2 is a prefix, e.g., 1.2.2-004. The exact syntax and definition of version-id's and version strings are described in Appendix A.

Section 6 describes how resources are downloaded and how the version information is associated with the resources.

3.5 Descriptor Information

The information element contains information intended to be consumed by the JNLP Client to integrate the application into the desktop, provide user feedback, etc. For example:

```
<information>
  <title>Cool App 1.0</title>
  <vendor>My Corporation</vendor>
  <description>Helps you keep cool</description>
  <description kind="tooltip">CoolApp</description>
  <homepage href="doc/index.html"/>
  <icon href="icon.gif"/>
  <offline-allowed/>
</information>
<information locale="da_DK">
  <description>Lidt for koldt?</description>
  <description kind="tooltip">Køligt</description>
</information>
```

locale attribute: The locales for which the information element should be used. Several locales can be specified, separated with spaces. Each locale is specified by a language identifier, a possibly country identifier, and possibly a variant[5]. The syntax is as follows:

```
locale ::= language [ "_" country [ "_" variant ] ]
```

An information element matches the current locale if i) the locale attribute is not specified or is empty, or ii) if one of the locales specified in the locale attribute matches the current locale. The rules for matching the current locale are as follows:

- If language, country, and variant are specified, then they must all match the current locale.
- If only language and country are specified, then they must match the language and country of the current locale.
- If only language is specified, then it must match the language of the current locale.

The match is case-insensitive.

The JNLP Client must search through the information elements in the order specified in the JNLP file. For each information element, it checks if the value specified in the locale attribute matches the current locale[6]. If a match is found, the values specified in that information element will be used, possibly overriding values found in previous information elements.

In the above example, the descriptions have been localized for the Danish locale, so these description values will be used whenever the current locale is matched by "da_DK". Since the information element for Danish includes values only for the descriptions, the values for all other elements (title, vendor,etc.) are taken from the information element without a locale attribute. For all other locales besides Danish, all values are taken from the information element with no locale attribute. Thus, the locale-independent information needs only to be specified once, in the information element without the locale attribute.

title element: The name of the application.

vendor element: The name of the vendor of the application.

5 Language codes are defined by ISO 639, and country codes by ISO 3166.
6 The current locale for a JNLP Client could, for example, be the one returned by `Locale.getDefault()`.

homepage element: Contains a single attribute, href, which is a URL locating the home page for the application. It can be used by the JNLP Client to point the user to a Web page where they can find more information about the application.

description element: A short statement about the application. Description elements are optional. The kind attribute defines how the description should be used, it can have one of the following values:

- *one-line*: If a reference to the application is going to appear in one row in a list or a table, this description will be used.

- *short*: If a reference to the application is going to be displayed in a situation where there is room for a paragraph, this description is used.

- *tooltip*: A description of the application intended to be used as a tooltip.

Only one description element of each kind can be specified. A description element without a kind is used as a default value. Thus, if a JNLP Client wants a description of kind *short*, and it is not specified in the JNLP file, then the text from the description without an attribute is used.

All descriptions contains plain text. No formatting, such as, e.g., HTML tags are supported.

icon element: The icon can be used by a JNLP Client to identify the application to the user.

The optional width and height attributes can be used to indicate the resolution of the images. Both are measured in pixels.

The optional depth attribute can be used to describe the color depth of the image.

The optional kind attribute can be used to indicate the use of the icon, such as default, selected, disabled, and rollover.

The optional size attribute can be used to specify the download size of the icon in bytes.

The JNLP Client may assume that a typical JNLP file will have at least an icon of 32x32 pixels in 256 colors of the default kind. The image file can be in either GIF or JPEG format. Its location is specified as described in Section 3.4, and it is downloaded using the protocols described in Section 6.

offline-allowed element: The optional offline-allowed element indicates if the application can work while the client system is disconnected from the network. The default is that an application only works if the client system is online.

This can be use by a JNLP Client to provide a better user experience. For example, the offline allowed/disallowed information can be communicated to the user, it can be used to prevent launching an application that is known not to work when the system is offline, or it can be completely ignored by the JNLP Client. An application cannot assume that it will never be launched offline, even if this element is not specified.

3.6 Application Descriptors

An application descriptor either describes an application or an Applet.

3.6.1 Application Descriptor for an Application

A JNLP file is an application descriptor if the application-desc element is specified.

The application-desc element contains all information needed to launch an application, given the resources described by the resources element. For example:

```
<application-desc main-class="com.example.MyMain">
  <argument>Arg1</argument>
  <argument>Arg2</argument>
</application-desc>
```

main-class attribute: The name of the class containing the `public static void main(String[])` method of the application. This attribute can be omitted if the main class can be found from the `Main-Class` manifest entry in the main JAR file. See Section 5.2.

argument element: Contains an ordered list of arguments for the application.

Section 5.2 describes how an application is launched.

3.6.2 Application Descriptor for an Applet

A JNLP file is an application descriptor for an Applet if the applet-desc element is specified.

The applet-desc element contains all information needed to launch an Applet, given the resources described by the resources elements. For example:

```
<applet-desc
   main-class="com.mysite.MyApplet"
   documentbase="index.html"
   name="MyApplet"
   width="500"
   height="300">
  <param name="Param1" value="Value1"/>
  <param name="Param2" value="Value2"/>
</applet-desc>
```

main-class attribute: Name of the main Applet class. This is the name of the main Applet class (e.g., com.mysite.MyApplet) , as opposed to the HTML <applet> tag's code attribute is a filename (e.g., MyApplet.class).

documentbase attribute: Documentbase for the Applet as a URL. This is available to the Applet through the `AppletContext`. The documentbase is provided explicitly since an Applet launched with a JNLP Client is not embedded in a Web page.

name attribute: Name of the Applet. This is available to the Applet through the `AppletContext`.

width attribute: Width of the Applet in pixels.

height attribute: Height of the Applet in pixels.

param element: Contains a parameter to the Applet. The name attribute contains the name of the parameter, and the value attribute contains the value. The parameters can be retrieved with the `Applet.getParameter` method.

The codebase for the Applet, available through the `java.applet.getCodebase` method, defaults to the value of the codebase attribute of the jnlp element. If no value is specified for that attribute, then the codebase is set to the URL of the JAR file containing the main Applet class.

Section 5.2 describes how an Applet is launched.

3.7 Extension Descriptors

An extension descriptor can either describe a component extension or an installer extension.

3.7.1 Component Extension

A JNLP file is a component extension if the component-desc element is specified. A component extension is typically used to factor out a set of resources that are shared between a large set applications. For example, this could be a toolkit for XML parsing. The following shows a sample JNLP fragment that specifies a component descriptor:

```
<jnlp>
  ...
  <resources>
     <!-- Resources defined by the component-desc -->
     <jar href="http://www.mysite.com/my-component/A.jar"/>
     ...
  </resources>
  <component-desc/>
</jnlp>
```

No j2se elements can be specified as part of the resources. Section 4 describes how these resources become part of the application that uses the extension.

An extension descriptor is downloaded using the extension download protocol described in Section 6.4.

3.7.2 Installer Extension

A JNLP file is an installer extension if the installer-desc element is specified. It describes an application that is executed only once, the first time the JNLP file is used on the local system. The following shows a sample JNLP fragment that specifies an installer descriptor:

```
<jnlp>
  ...
  <resources>
     <!-- Resources used for installer -->
     <jar href="http://www.mysite.com/my-installer/installer.jar"/>
     ...
  </resources>
  <installer-desc main-class="com.mysite.installer.Main"/>
</jnlp>
```

main-class attribute: The name of the class containing the `public static void main(String[])` method of an installer/uninstaller for this extension. This attribute can be omitted if the main class can be found from the `Main-Class` manifest entry in the main JAR file. This is described in detail in Section 5.2.

The installer extension is intended to install platform-specific native code that requires a more complicated setup than simply loading a native library into the JVM, such as installing a JRE or device driver. The installer executed by the JNLP Client must be a Java Technology-based application. Note that this does not limit the kind of code that can be installed or executed. For example, the installer could be a thin wrapper that executes a traditional native installer, executes a shell script, or unzips a ZIP file with native code onto the disk.

The installer communicates with the JNLP Client using the `ExtensionInstallerService` (see section 7.8 for details). Using this service, the installer informs the JNLP Client what native libraries should be loaded into the JVM when the extension is used, or, in the case of a JRE installer, inform the JNLP Client how the installed JRE can be launched.

Installers should avoid having to reboot the client machine if at all possible. While some JNLP Clients may be able to continue with the installation/launch after a reboot, this ability is not required.

4 Application Resources

The resources element is used to specify all the resources, such as Java class files, native libraries, and system properties, that are part of an application.

4.1 Overview

The resources element has 6 different possible subelements: jar, nativelib, j2se, property, package, and extension. These are all described in detail in this section.

A resources definition can be restricted to a specific operating system, architecture, or locale using the os, arch, and locale attributes. For example:

```
<resources>
  <j2se version="1.2"/>
  <jar href="lib/myjar.jar" version="1.2"/>
  <extension
    name="coolaudio" version="1.0"
    href="http://www.mysite.com/ext/coolaudio">
    <part name="mp3"/>
  </extension>
  <property name="key1" value="value1"/>
  <property name="key2" value="value2"/>
</resources>
<resources os="SunOS">
  <jar href="lib/motif-plaf.jar"/>
</resources>
```

os attribute: Specifies the operating system for which the resources element should be considered. If the value is a prefix of the `os.name` system property, then the resources element can be used. If the attribute is not specified, it matches all operating systems.

arch attribute: Specifies the architecture for which the resources element should be considered. If the value is a prefix of the `os.arch` system property, then the resources element can be used. If the attribute is not specified, it matches all architectures.

locale attribute: Specifies that the resources element is locale-dependent. If specified, the resources element should only be used if the default locale for the JNLP Client matches one of the specified locales. If the attribute is not specified, then it matches all locales. The locale is specified and matched as described for the locale attribute of the information element (see Section 3.5).

For the os, arch, and locale attributes several keys can be specified separated with spaces. A space that is part of a key must be preceded with a backslash (\). For example, "Windows\ 95 Windows\ 98" specifies the two keys "Windows 95" and "Windows 98".

4.2 Setting System Properties

The property element defines a system property that will be available through the `System.getProperty` and `System.getProperties` methods. Is has two required attributes: name and value. For example:

```
<property name="key1" value="value1"/>
```

Properties must be processed in the order specified in the JNLP file. Thus, if two properties define different values for the same property, then the last value specified in the JNLP file is used. For example, given the following two declarations, n the given order:

```
<property name="key" value="overwritten"/>
<property name="key" value="used"/>
```

Then the property *key* will have the value *used*.

4.3 Specifying Code Resources

A JNLP file may have two kinds of code resources:

- A jar element specifies a JAR file that is part of the application's classpath. The JAR file will be loaded into the JVM using a `ClassLoader` object. The JAR file will typically contain Java classes that contain the code for the particular application, but can also contain other resources, such as icons and configuration files, that are available through the `getResource` mechanism.

- A nativelib element specifies a JAR file that contains native libraries[7]. The JNLP Client must ensure that each file entry in the root directory of the JAR file (i.e., /) can be loaded into the running process by the `System.loadLibrary` method. It is up to the launched application to actually cause the loading of the library (i.e., by calling `System.loadLibrary`). Each entry must contain a platform-dependent shared library with the correct naming convention, e.g., *.dll on Windows, or lib*.so on Solaris.

The following JNLP file fragment shows how jar and nativelib elements are used. Notice that native libraries would typically be included in a resources element that is guarded against a particular operating system and architecture.

```
<resources>
  <jar href="lib/app.jar" version="3.2" main="true"/>
</resources>
<resources os="Windows"/>
  <nativelib href="lib/windows/corelibs.jar"/>
</resources>
<resources os="SunOS" arch="SPARC">
  <nativelib href="lib/solaris/corelibs.jar"/>
</resources>
```

The href attribute is the HTTP URL of a JAR file that the application depends on. The optional version attribute describes the required version, as described in Section 3.4. Section 6 describes how JAR files are downloaded. An optional size attribute can be used to indicate the download size of the JAR file in bytes.

The jar element has an main attribute (as shown above) that is used to indicate which JAR file contains the main class of the Application/Applet (or Installer for an extension). There must be at most one jar element in a JNLP file that is specified as *main*. If no jar element is specified as main, then the first jar element will be considered the main JAR file.

7 A native library is also called a DLL (dynamic linked library) on Windows and a shared object file (.so) on UNIX systems.

4.3.1 Use of Manifest Files

A JNLP Client ignores all manifest entries in a JAR file specified with the jar element, except the following:

- The manifest entries used to sign a JAR file are recognized and validated.
- The `Main-Class` entry in the JAR file specified as *main* is used to determine the main class of an application (if it is not specified explicitly in the JNLP file).
- The manifest entries used to seal a package are recognized, and the sealing of packages are verified according to the Extension Mechanism Architecture[8]. These are the *name* and *sealed* entries.
- The following manifest entries described by the Optional Package Versioning documentation[9]: *Extension-Name*, *Specification-Vendor*, *Specification-Version*, *Implementation-Vendor-Id*, *Implementation-Vendor*, and *Implementation-Version* are recognized and will be available through the `java.lang.Package` class. They are otherwise not used by a JNLP Client.

For a JAR file containing native libraries, i.e., specified with the nativelib element, all manifest entries are ignored except the entries used to sign the JAR file.

4.4 Parts and Lazy Downloads

By default, the jar and nativelib resources must be downloaded eagerly, i.e., they are downloaded and available locally to the JVM running the application before the application is launched. The jar and nativelib elements also allow a resource to be specified as *lazy*. This means that the resource does not necessarily need to be downloaded onto the client system before the application is launched. However, a JNLP Client is always allowed to eagerly download all resources if it chooses.

The download attribute is used to control whether a resource is downloaded eagerly or lazily. For example,

```
<jar href="sound.jar" download="lazy"/>
<nativelib href="native-sound.jar" download="eager"/>
```

The default value for the download attribute is *eager*.

From a functional point of view (i.e., assuming an infinitely fast and reliable network connection), it makes no difference if a JAR file is specified as lazy or eager. The JNLP Client must dynamically download and link in lazily-downloaded JAR files during the execution of the application when they are needed.

The Java Virtual Machine (JVM) will make requests to the application's classloader when it needs to resolve a class that is not currently loaded into the current JVM. The JNLP Client must make sure to intercept these requests (e.g., by installing its own classloader), and if there are JAR files specified in the JNLP file that are currently not loaded into the JVM, then the JNLP Client must download them and load them into the application's JVM.

The jar and nativelib elements also contain a part attribute that can be used to group resources together

8 See http://java.sun.com/j2se/1.3/docs/guide/extensions/spec.html
9 See http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html

so they will be downloaded at the same time. Whenever a jar or nativelib resource with a non-empty part attribute is being downloaded, then the JNLP Client must ensure that all other resources that have the same value in the part attribute are also downloaded.

Resources must be downloaded for the following events:

1. All resources specified as non-lazy must be downloaded before the application is launched: This might trigger download of resources that have the same part name.

2. If the JVM triggers a resource to be downloaded through a classloader request, then the classloader must not return until the JAR file containing the requested class is downloaded *and* all resources (jar or nativelib) that have the same (non-empty) value in the part attribute have been downloaded.

3. JAR files and parts can also be requested to be downloaded explicitly by the application program using the JNLP API. This is described in Section 5.3.

The part names are local to each JNLP file. The JNLP file for the application might define a part named *sound-support*, and an extension that is being used by the JNLP descriptor might also define a part named *sound-support.* These are considered two different part names. Thus, the scope of a part name is the JNLP file.

Native libraries, specified with the nativelib element, can also be downloaded lazily and loaded into the JVM while the application is running. A JVM does not generate requests to the classloader when a native library is missing. Thus, the only way a native library can be triggered to be downloaded and loaded into the JVM process is by using the part attribute. For example, when the Java classes that implement the native wrappers for the native libraries are downloaded, that can also trigger the download of the native library.

The following JNLP fragment shows an example of the use of the jar and nativelib element for lazy download of resources:

```
<resources>
   <jar href="sound.jar"
            part="sound" download="lazy"/>
</resources>
<resources os="Windows"/>
   <nativelib href="sound-native-win.jar"
            part="sound" download="lazy"/>
</resources>
<resources os="SunOS"/>
   <nativelib href="sound-native-solaris.jar"
            part="sound" download="lazy"/>
</resources>
```

The `sound.jar` file does not need to be downloaded before the application is launched, because it is specified as a *lazy* download. The native code for the sound library is also specified as lazy and is also in the *sound* part. The download of the `sound.jar` file will trigger the download and loading of the platform-dependent native code, i.e., either `sound-native-win.jar` on Windows, or `sound-native-solaris.jar` on Solaris.

4.5 PACKAGE ELEMENT

The package element can be used to indicate to the JNLP Client which packages are implemented in which JAR files. The name attribute specifies a package name, and the part attribute specifies which part must be downloaded in order to load that particular package into the JVM. The package element can take several forms:

```
<package name="com.mysite.Main" part="xyz"/>
```

Specifies that the class `com.mysite.Main` can be found in the part named *xyz*.

```
<package name="com.mysite.sound.*" part="abc"/>
```

Specifies that classes in the `com.mysite.sound` package can be found in the part named *abc*. The use of the "*" is similar to the `import` statement in the Java Programming Language. Thus, it is not a general purpose wildcard. Finally, the recursive attribute can be used to specify sub-packages as well.

```
<package name="com.mysite.sound.*" part="stu" recursive="true"/>
```

Specifies that all packages that have "`com.mysite.sound.`" as a prefix can be found in the part named *stu*. The recursive attribute only has an effect when used with a package name, i.e., a name that ends with ".*".

The package element only makes sense to use with lazily-downloaded resources, since all other resources will already be available to the JVM. Thus, it will already know what packages are implemented in those JAR files. However, it can direct the JNLP Client to download the right lazy JAR resources, instead of having to download each individual resource one at a time to check.

4.6 JAVA RUNTIME ENVIRONMENT

The j2se element (subelement of resources) specifies what Java 2 SE Runtime Environment (JRE) versions an application is supported on, as well as standard parameters to the Java Virtual Machine. Several JREs can be specified, which indicates a prioritized list of the supported JREs, with the most preferred version first. For example,

```
<j2se version="1.3" initial-heap-size="64m"/>
<j2se version="1.2">
  <resources> ... </resources>
</j2se>
```

version attribute: Describes supported versions of the JRE. The exact syntax and interpretation of the version string is described in Section 4.6.1.

initial-heap-size attribute: Indicates the initial size of the Java heap. The modifiers m and k can be used for megabytes and kilobytes, respectively. For example, "128m" will be the same as specifying "134217728" (128*1024*1024). The modifiers are not case-sensitive.

max-heap-size attribute: Indicates the maximum size of the Java heap. The modifiers m and k can be used for megabytes and kilobytes, respectively. For example, "128m" will be the same as specifying "134217728" (128*1024*1024). The modifiers are not case-sensitive.

resources element: A j2se element can contain nested resources elements. If the JRE specified in the

enclosing j2se element is chosen by the JNLP Client, then the resources specified in the nested resources also becomes part of the applications resources, otherwise they are ignored. Any j2se element in this resource element is ignored.

4.6.1 Java Runtime Environment Version Specification

A JRE can be specified in two ways in the JNLP file. It can be specified in a vendor-independent manner by referring to a particular platform version of the Java 2 platform, or it can be specified by using a product version of a particular JRE vendor's implementation.

Definition: Platform version

It is the version of a particular revision of the Java 2 platform. A platform version describes a particular set of APIs (classes and interfaces), semantics, and syntax of the Java 2 platform.

The version id is of the form 'x.y'. Occasionally, dot-dot releases can be released, like 'x.y.z'. This would typically be in response to a security update. Current versions (as of this writing) are 1.2 and 1.3.

The platform version of a JRE can be determined by examining the `java.specification.version` system property.

Definition: Product version

It is the version of a particular implementation of the Java 2 platform. The product version is vendor-specific. A product implements a specific platform version. The product version and platform versions are not necessarily related.

The product version can be found by examining the `java.version` system property[10].

If no href attribute is specified, the version string refers to a platform version of the Java 2 platform. For example,

```
<j2se version="1.2">
```

The JNLP Client can select any JRE implementation that implements this particular revision (as given by the `java.specification.version` system property).

If an href attribute is specified, a vendor-specific JRE is requested. A specific JRE implementation is uniquely named using a URL and a product version. For example[11]:

```
<j2se href="http://java.sun.com/products/j2se"
      version="1.2.2+"/>
```

The product version of a JRE implementation can be extracted from the `java.version` system property. Each JRE vendor will be responsible for providing the unique URL that names their particular

10 This practice is followed starting from Java 2 SE JRE 1.3.0 for Sun's implementations. The java.version system property is "1.2.2" for several different product versions, such as "1.2.2-w" (final release) and "1.2.2-001" (patch release). As a workaround, to determine the actual product version, use `java -fullversion`.

11 The above URL is only used as an example. The exact URL for naming Sun's JREs is likely to be different.

implementations.

As a general rule, a product version should typically be specified using a prefix or a greater than notation, i.e., be postfixed with either a plus (+) or a star (*). This will obviously put less download strain on the client, since a greater set of JVM implementations can be used. However, more importantly, a specific product version might be obsolete due to, e.g., security problem. If this happens, the user will be unable to run the particular application, if the JNLP file does not specify that a later version with the particular problem fixed can be used.

Section 6.4 explains how the URL can also be used by the JNLP Client to download and install a JRE, if it is not already present on the local machine.

4.6.2 Selecting what JRE to Use

The JNLP Client can choose any of the JRE combinations that are specified in the JNLP file. Consider the following JNLP file fragment:

```
<resources os="Windows" arch="x86">
   <j2se href="http://java.sun.com/..." version="1.2.2-w">
     <resources>
       <jar href="http://www.mysite.com/app122.jar"/>
       <extension name="MyExt" href="http:..." version="3.4"/>
       <property name="CheckThis" value="Wow"/>
     </resources>
   </j2se>
</resources>
<resources>
  <j2se version="1.3" initial-heap-size="64m"/>
</resources>
```

The following two combinations would be legal to launch the application on:

- Sun's Java 2 SE JRE, version 1.2.2-w given that you are running on the Windows operating system and the x86 architecture.

- Any JRE compatible with the Java 2 platform, version 1.3. No special extensions are needed and no operating system nor architecture constraints are specified.

If any of the specified JRE/extensions combinations are already installed on the client machine, then the one listed earliest in the list should be used. If none are installed, the JNLP client may pick which one to download and install. In general, JRE/extensions combinations that appear earlier in the list should be preferred over ones that appear later; however, the JNLP Client may take other factors into account, such as minimizing download times. The download and installation procedure for JREs and extensions is described in Section 6.

JRE-specific resources can be specified by including a nested resources element inside a j2se element. The nested resources elements are all ignored, except the one in the j2se element that is used to launch the application. Thus, in the above example, if the application is launched using Sun's 1.2.2-w JRE, then the property `CheckThis` will be set to `Wow`, and the `app122.jar` JAR file will be part of the application's resources, as well as the MyExt extension.

4.7 EXTENSION RESOURCES

Extension descriptors can be included as part of the resources for an application (or extension) by using the extension element. For example:

```
<resources>
  ...
  <extension name="Sound" version="1.0"
    href="http://www.myext.com/servlet/ext/sound-extension.jnlp">
    <ext-download ext-part="MIDI"/>
    <ext-download ext-part="MP3"
                  download="lazy"
                  part="MP3Player"/>
  </extension>
  ...
</resources>
```

The extension element contains three attributes: name, version, and href. The href and the optional version attributes uniquely identify the extension. This href typically does not point to a file but to, e.g., a servlet that understands the extension download protocol. The extension itself is described by an extension descriptor, i.e., a JNLP File. How the extension descriptor is downloaded is described in Section 6. The name attribute can be used by the JNLP Client to inform the user about the particular extension that is required, while the extension descriptor (i.e., the JNLP file) is being downloaded.

The inclusion of an extension element in a resources element has the following effect:

- If it points to a component extension (i.e., a JNLP file with a component-desc element), then the resources described in the resources element in that JNLP file become part of the application's resources. The included resources will have the permissions specified in the component extension.

- If the extension points to an extension installer (i.e., a JNLP file with an installer-desc element), then the installer application will be executed, if it has not already been executed on the local machine. This is described in detail in Section 5.2.

For an extension element that points to component extension, it can also be specified when the different parts of the component extension should be downloaded. This is done using the ext-download subelements. In the above example, the extension part *MIDI* is specified to be downloaded eagerly, and the part in the extension descriptor named *MP3* must be downloaded at the same time as the part named *MP3Player* in the JNLP file containing the extension element. Note that a JNLP Client is always allowed to eagerly download all parts if it chooses. This is all described in more detail in Section 4.4.

Given the example above, and if the extension descriptor for the *Sound* extension contains the following component extension:

```
<jnlp>
   ...
   <resources>
       <jar href="http://www.myext.com/lib/midi.jar" part="MIDI"
        download="lazy"/>
       <jar href="http://www.myext.com/lib/mp3.jar" part="MP3"
        download="lazy"/>
    </resources>
  <component-desc/>
</jnlp>
```

Then the extension element in the previous example would be, in effect[12], replaced with the following two JAR files specified in the extension descriptor:

```
  <resources>
     ...
     <jar href="http://www.myext.com/lib/midi.jar" download="eager"/>
     <jar href="http://www.myext.com/lib/mp3.jar" download="lazy"
          part="MP3Player"/>
     ...
  </resources>
```

12 Except for permissions, if the component extension and application descriptor specified different permissions in the security element.

5 Launching and Application Environment

This section describes the steps a JNLP Client must take to download and launch an application, Applet, library, or extension installer/uninstaller, and the environments these applications will be run in.

5.1 Launch Sequence

A JNLP Client performs the following steps to launch an application:

1. Retrieve a JNLP file. The JNLP file might, for example, be supplied as an argument, looked up in a cache managed by the JNLP Client, or downloaded from a URL.

   The JNLP file can either be an application descriptor or an extension descriptor. For an application-descriptor, it will describe how to launch an Application or Applet. For an extension descriptor, it will describe how to launch an installer/uninstaller for the extension.

2. Parse the JNLP file.

   The JNLP Client must abort the launch if the JNLP file could not be parsed, due to, e.g., syntax errors, missing fields, or fields with illegal values.

   There must always be a title and vendor specified based on the current locale.

3. Determine the right JRE to use. This might require downloading and installing the appropriate JRE. (See Section 4.6.2.)

4. Download extension descriptors for all the extensions used in the JNLP file. This step continues recursively with the extensions specified in the downloaded extension descriptors. Section 6 specifies how extension descriptors are downloaded and cached.

5. Run the installer for any required extension for which the installer has not yet been run. This is explained in Section 5.6.

6. Download all *eager* JAR files (jar and nativelib elements) specified in the JNLP file from Step 1 (i.e., defined by the resources subelement of the jnlp element), and recursively defined by extension descriptors. Section 4 describes which resources are eager.

7. Verify the signing and security requirements.

   This is explained in detail in Section 5.3.

8. Setup the JNLP Services.

This is explained in detail in Section 7.

9. Launch the application/Applet/installer/uninstaller.

   This is explained in detail in Section 5.2.

5.1.1 Launch Offline

An application can be launched offline if the launch sequence described above can be completed without the need to download any resources, i.e., all JAR resources in Step 6 must already be downloaded and cached, all JNLP files must be cached locally, and the JRE must be available locally.

The application itself might not support offline operation. See the description of the offline-allowed element in Section 3.5.

5.2 Launching Details

5.2.1 Launching an Application

If the JNLP file contains the application-desc element, then an application must be launched.

The main class for the application is by default determined by the main-class attribute of the application-desc element. If this is not specified, then the `Main-Class` manifest entry for the main JAR file is used. If neither is specified, then the launch must be aborted.

The application is launched by invoking the `static public void main(String[] argv)` method of the main class. The `argv` argument is constructed from the argument elements of the application-descriptor element.

5.2.2 Launching an Applet

If the JNLP file contains the applet-desc element, then an Applet must be launched.

The main class for the Applet is by determined by the main-class attribute of the applet-desc elements.

To launch the Applet, this will require setting up an Applet container, instantiating the Applet, and invoking the `init` and `start` methods.

5.2.3 Launching an Extension Installer/Uninstaller

If the JNLP file contains the installer-desc element, then the JNLP file defines an extension installer/uninstaller. This section describes how the installer/uninstaller gets invoked.

The main class for the extension installer/uninstaller is by default determined by the main-class attribute of the installer-desc element. If this is not specified, then the `Main-Class` manifest entry for the main JAR file is used. If neither is specified, then the the launch must be aborted.

The extension installer must be executed before the application that depends on it is launched. Furthermore, the installer must only be run once, i.e., the first time the installer extension is downloaded. However, if the uninstaller is later executed, the JNLP Client must ensure that the extension is installed again before it is used the next time.

For installation, the JNLP Client must invoke the `public static void main(String[])` method in the specified class with the `String` array `{"install"}` as an argument.

For uninstallation, the JNLP Client must invoke the `public static void`

`main(String[])` method in the specified class with the `String` array `{"uninstall"}` as an argument.

An installer is launched through an extension descriptor just like an application is launched through an application descriptor. Thus, an extension installer is by default run in a restricted environment. The all-permissions element can be specified in the security section to request unrestricted access. An extension with an installer will typically need to be signed, so the installer can gain access to the local file system.

5.3 Application Environment

An application launched with a JNLP Client must be run in an environment according to the specification below.

At the core of the environment is the Java 2 SE platform standard environment, i.e., the environment provided by the Java 2 SE JRE for all Java Technology-based applications. On top of this core environment, this specification defines the following additional requirements:

- A preconfigured set of proxies for HTTP, so basic communication with HTTP through the `java.net.URL` class works correctly.

- A restricted execution environment (aka. sandbox) for untrusted applications, and two execution environments for trusted applications. The trusted environments are the all-permissions and j2ee-application-client environments.

- A basic set of services that are available through the *javax.jnlp* package.

- The ability to download application resources (such as JAR files) lazily as the application executes. This download will typically be initiated based on a class resolution request in the JVM.

- Validating signing of the JAR files.

The execution environments are described in more detail in the following sections.

5.4 Signed Applications

The signing infrastructure for JNLP is built on top of the existing signing infrastructure for the Java 2 Platform. The Java 2 Platform supports signed JAR files. A JAR file can be signed and verified using, e.g., the standard `jarsigner` tool from the Java 2 SDK.

An application launched by a JNLP Client is considered to be signed, if and only if:

- All the JAR files are signed (both for jar elements and nativelib elements) and can be verified. A JAR file is signed if the signature covers all the entries in the JAR file[13]. A single certificate must be used to sign each JAR file.

- If a signed version of the JNLP file exists, then it must be verified, and it must match the JNLP file used to launch the application. This is described in the following subsection.

- The same certificate is used to sign all JAR files (jar and *nativelib* elements) that are part of a single

13 Strictly speaking, the manifest and the signature files are not signed, since they contain the signing information.

JNLP file. This simplifies user management since only one certificate needs to be presented to the user during a launch per JNLP file (and hardly restricts the signing process in practice).

- The certificate used for signing the JAR files and JNLP file (if signed) must be verified against a set of trusted root certificates.

How the set of trusted root certificates are obtained depends on the particular JNLP Client implementation. Typically, a JNLP Client will be shipped with a set of trusted root certificates for all the major Certificate Authorities (CAs).

The JNLP Client must check a JAR file for signing information before it is used, i.e., before a class file or another resource is retrieved from it. If a JAR file is signed and the digital signature does not verify, the application must be aborted. For a lazily downloaded JAR file, i.e., a JAR file that is downloaded after the application is launched, this might require aborting an already-running application.

5.4.1 Signing of JNLP Files

A JNLP file can optionally be signed. A JNLP Client must check if a signed version of the JNLP file exists, and if so, verify that it matches the JNLP file that is used to launch the application. If it does not match, then the launch must be aborted. If no signed JNLP file exists, then the JNLP file is not signed, and no check needs to be performed.

A JNLP file is signed by including a copy of it in the signed main JAR file. The copy must match the JNLP file used to launch the application. The signed copy must be named: `JNLP-INF/APPLICATION.JNLP`. The `APPLICATION.JNLP` filename should be generated in upper case, but should be recognized in any case.

The signed JNLP file must be compared byte-wise against the JNLP file used to launch the application. If the two byte streams are identical, then the verification succeeds, otherwise it fails.

As described above, a JNLP file is not required to be signed in order for an application to be signed. This is similar to the behavior of Applets, where the Applet tags in the HTML pages are not signed, even when granting unrestricted access to the Applet.

5.5 Untrusted Environment

All applications are by default run in an untrusted or restricted environment by a JNLP Client. The restricted environment is similar to the well-known Applet sandbox, and is designed so untrusted applications are prevented from intentionally or unintentionally harming the local system. For example, the restricted environment limits access to local disk and the network.

When run in the restricted execution environment, the following restrictions must be enforced on the application:

Single download host: All JAR files specified in the resources elements of the JNLP file must be downloaded from the same host, the *download host*.

Native libraries: No nativelib elements can be used.

Security Manager: The application must be run with a `SecurityManager` installed. The

following table list the exact set of permissions[14] that must be granted to the application's resources:

Security Permissions	Target	Action
`java.net.SocketPermission`	localhost:1024-	`listen`
`java.net.SocketPermission`	<Download Host> (see above)	`connect, accept`
`java.util.PropertyPermission`	java.version	`read`
`java.util.PropertyPermission`	java.vendor	`read`
`java.util.PropertyPermission`	java.vendor.url	`read`
`java.util.PropertyPermission`	java.class.version	`read`
`java.util.PropertyPermission`	os.name	`read`
`java.util.PropertyPermission`	os.version	`read`
`java.util.PropertyPermission`	os.arch	`read`
`java.util.PropertyPermission`	file.separator	`read`
`java.util.PropertyPermission`	path.separator	`read`
`java.util.PropertyPermission`	line.separator	`read`
`java.util.PropertyPermission`	java.specification.version	`read`
`java.util.PropertyPermission`	java.specification.vendor	`read`
`java.util.PropertyPermission`	java.specification.name	`read`
`java.util.PropertyPermission`	java.vm.specification.vendor	`read`
`java.util.PropertyPermission`	java.vm.specification.name	`read`
`java.util.PropertyPermission`	java.vm.version	`read`
`java.util.PropertyPermission`	java.vm.vendor	`read`
`java.util.PropertyPermission`	java.vm.name	`read`
`java.lang.RuntimePermission`	exitVM	
`java.lang.RuntimePermission`	stopThread	
`java.awt.AWTPermission`	showWindowWithoutWarningBanner	
`java.awt.AWTPermission`	accessEventQueue (see below for details)	

Properties: A application has read and write access to all the properties specified in the JNLP file. Properties specified in the JNLP file must overwrite the default value of the above properties.

Event Queue Access: A JNLP Client must ensure that an application only has access to its own event queue. For example, if a JNLP Client restricts access to the clipboard in the `ClipboardService` by displaying a security advisor dialog, then the JNLP Client must ensure that this dialog is on an event queue that the launched application does *not* have access to. This ensures that the application cannot programmatically circumvent the security measures implemented by a JNLP Client.

14 The J2SE sercurity permissions are fully described in `http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html`.)

Permission to the event queue can be granted by a JNLP Client by overwriting the `checkAwtEventQueueAccess` on the `SecurityManager` object, instead of explicitly adding the `java.awt.AWTPermission("accessEventQueue")` permission.

Extensions: The JNLP file can request extensions and JREs from any host. An application cannot make a socket connection back to any of the hosts where JREs or extensions are downloaded from (unless it happens to be the same as for the JAR files). Extensions requested from hosts other than the one that the JAR files were downloaded from must be signed and trusted as per Section 5.4.

This environment is a superset of the Applet sandbox. Since only one application is run per JVM for an application launched by a JNLP Client, the JNLP sandbox does not have to restrict access to, e.g., `System.exit`.

5.6 Trusted Environments

This specification specifies two trusted environments, the all-permissions environment and an environment that meets the security specifications of the J2EE Application Client environment. Both of these environments provide unrestricted access to the network and local disk. Thus, an application can intentionally or unintentionally harm the local system. An application must only be launched if it is trusted.

The security element in the JNLP file is used to request the trusted environments:

All Permissions	J2EE Application Client Permissions

`<security>`
`  <all-permissions/>`
`</security>`
	`<security>`
	`  <j2ee-application-client-permissions/>`
	`</security>`

The following requirements must be satisfied before a JNLP Client can grant an application these access rights:

1. The application is signed.

2. The user and/or the JNLP Client trusts the certificate that is used to sign the application.

How a JNLP Client decides to trust a certificate is dependent on the particular implementation. Typically, a JNLP Client would prompt the user to make a decision on whether to launch the application or not. The decision can be based on the information stored in the certificate. The decision can be cached, so the accept action is only required the first time the application is launched.

The application must be run with a `SecurityManager` installed. The following table lists the exact set of permissions that must be granted to the application's resources:

Security Permissions	Target	Action
All Permissions Environement		
`java.security.AllPermission`		
J2EE Application Client Environment		

Security Permissions	Target	Action
`java.awt.AWTPermission`	accessClipboard	
`java.awt.AWTPermission`	accessEventQueue	
`java.awt.AWTPermission`	showWindowWithoutWarningBanner	
`java.lang.RuntimePermission`	exitVM	
`java.lang.RuntimePermission`	loadLibrary	
`java.lang.RuntimePermission`	queuePrintJob	
`java.net.SocketPermission`	*	connect
`java.net.SocketPermission`	localhost:1024-	accept, listen
`java.io.FilePermission`	*	read, write
`java.util.PropertyPermission`	*	read

An application running with all-permissions can create its own classloader to, e.g., install code downloaded from the network. Note that when using custom classloader, the application might circumvent the caching mechanisms provided by the JNLP Client and thereby degrade the performance of the application.

5.7 Execution Environment for Component Extensions

All the JAR files specified in the resources elements for a component extension become part of the application's resources. JAR files for an extension must execute with the set of permissions specified in the extension descriptor, which is not necessarily the same set as the application.

By default, the component extension resources are executed in the untrusted execution environment. However, by using the security element, either of the trusted environments can be requested. Just as for an application, the resources must be signed and the user must trust the extension before it can be run in a trusted environment. An extension that contains native code will always need to request trusted access to the system.

6 Downloading and Caching of Resources

The JNLP Client can download four different kind of resources from the Web: JAR files, images, extensions, and JNLP files. This sections describes how they are downloaded, and how they can be cached on the client system.

JNLP defines three different download protocols, all based on HTTP GET requests:

- A basic download protocol which does not require special Web server support. This can be used for JAR files, images, and JNLP files.
- A version-based download protocol that allows greater control over which resources are downloaded and which supports incremental updating of JAR files. This can be used for JAR files and images.
- An extension download protocol which is an addition to the above protocols to include platform-specific information. This is used for JNLP files containing extension descriptors.

Resources are named uniquely in a JNLP file using the href and version attributes. The following table summarize the different elements that refer to external resources and which download protocols they support.

Element	href attribute	version attribute	Supported Protocols
jnlp	Optional	n/a	Basic
icon	Required	Optional	Basic, Version-based
jar	Required	Optional	Basic, Version-based
nativelib	Required	Optional	Basic, Version-based
extension	Required	Optional	Basic, Extension
j2se	Optional	Required	Extension

6.1 HTTP Format

The HTTP protocol is used to transfer all information between the Web server and the JNLP Client. The following describes the common request and response format used by all download protocols.

6.1.1 Request

Each request consists of an HTTP GET request to the URL for the given resource, and potentially a set of arguments, passed to the Web server in the query string of the URL. The syntax for a request is:

```
request   ::= href [ "?" arguments ]
arguments ::= key "=" value ( "&" key "=" value ) *
```

The *href* in the request is the exact URL for the given resource as specified in the JNLP file. The JNLP Client must not encode nor decode this part of the request. The key and value elements are encoded using the default encoding for parameters passed in URLs. To convert a key/value, each character is examined in turn:

- The ASCII characters 'a' through 'z', 'A' through 'Z', '0' through '9', '.', '*', '-', and '_' can remain the same.
- The space character ' ' is converted into a plus sign '+'.
- All other characters are converted into the 3-character string "%xy", where xy is the two-digit hexadecimal representation of the lower 8-bits of the character.

The `java.net.URLEncoder` in the Java 2 SE platform implements this conversion.

6.1.2 Response

If the HTTP response status code is different than 200 OK, then the request failed and the JNLP Client must handle this as an error.

If the HTTP request succeeded, JNLP Client must examine the MIME type of the HTTP response to figure out the kind of resource/response returned.

The following table gives an overview of the possible return MIME types and for which download protocols and elements they are used. The description of the individual protocols elaborates on this.

Return MIME type	Elements	Download Protocol
image/jpeg	icon	Basic, Version-based
image/gif	icon	Basic, Version-based
application/x-java-archive	jar, nativelib	Basic, Version-based
application/x-java-archive-diff	jar, nativelib	Version-based
application/x-java-jnlp-file	jnlp, extension, j2se	Basic, Extension
application/x-java-jnlp-error	All	All

If the *application/x-java-jnlp-error* MIME type is returned, the request failed. The response must be a single line that contains the numeric status code, followed by a space, followed by a textual explanation. The following status codes are defined:

Error Code	Download Protocol	Description
10	All	Could not locate resource
11	All	Could not locate requested version
20	Extension	Unsupported operating system
21	Extension	Unsupported architecture
22	Extension	Unsupported locale
23	Extension	Unsupported JRE version
99	All	Unknown error

The description returned from the Web server does not necessarily need to match the above descriptions.

The *10 Could not locate resource* is included for completeness. Typically, a Web server will use the *404 Not Found* HTTP status code to convey this information.

An unmodified Web server can be used with the basic protocol. It will never return the *10 Could not locate resource* error code. It will instead return the 404 HTTP status code.

6.2 Basic Download Protocol

The basic download protocol is used to download resources without any version information, i.e., where the version attribute is not specified. A resource is downloaded with an HTTP GET request to the Web server. For example, given the following jar element:

```
<jar href="http://www.mysite.com/c.jar"/>
```

then the JNLP Client must issue the following HTTP GET request:

```
http://www.mysite.com/c.jar
```

to retrieve the JAR file.

The JNLP Client must examine the HTTP response status code and MIME type to determine if the result was successful. The valid responses are described in Section 6.1.2

6.3 Version-based Download Protocol

For the version-based download protocol, all resources are uniquely identified by a URL/version-id pair. Thus, a JNLP Client can at any given time request a specific version of a resource located at a specific URL.

The JNLP Client issues an HTTP GET request that includes the specific version of the resource that it needs. The request includes the field `version-id`, which specifies the requested version. For example, given the following jar element:

```
<jar href="http://www.mysite.com/b.jar" version="2.3+"/>
```

then the JNLP Client must issue the following HTTP GET request[15]:

```
http://www.mysite.com/c.jar?version-id=2.3%2B
```

The JNLP Client must examine the HTTP response status code and MIME type to determine if the result was successful. The valid responses are described in Section 6.1.2. For the above jar element, the *application/x-java-archive-diff* MIME type cannot be returned. It can only be returned for incremental requests.

The version string used in the request is not necessarily exact, e.g., 2.3+. The Web server must specify the exact version-id of the resource that is returned in the response by setting the HTTP header field: `x-java-jnlp-version-id`. The exact version returned must be one that matches the requested version string.

15 The plus sign (+) in the version string is converted into the %2B given the standard encoding for arguments in URLs

6.3.1 Incremental Updates for JAR files

JNLP allows incremental updates to be applied to JAR files. Typically, downloading an incremental update will be much faster than downloading the new version. Incremental updates are distributed in the form of a JARDiff file, which are described in Appendix B.

If the JNLP Client has a previous version of a given JAR file already cached, e.g., version 2.2, then this fact can be specified in the request. The Web server can then potentially provide an incremental update that can be applied to the existing file, instead of returning the contents of the new file.

An incremental update is enabled by providing information about the version that is already cached by the JNLP Client in the HTTP request. The field `current-version-id` is used to specify the existing local version. For example:

```
http://www.mysite.com/c.jar?version-id=2.3%2B&current-version-id=2.2
```

The `current-version-id` must always be exact. If several versions of a given resource are in the cache, then the highest version-id that is lower than the requested version should be used. The Web server is not required to return an incremental update, but could just return the requested JAR file.

The returned contents of the response are the same as for the request without the `current-version-id` field, except that a JARDiff file might be returned. In that case, the response MIME type must be *application/x-java-archive-diff*.

6.4 Extension Download Protocol

An extension can either be named with a URL or a URL/version-id pair. If only a URL is specified, the basic download protocol is used to download the extension descriptor (i.e., the JNLP file). If both a URL and a version string are specified, the version-based download protocol plus a set of additional fields are used. The extra fields allow the Web server to return different extension descriptors for different platforms. The extension download protocol is also used to download a JRE.

The additional fields in the request are:

Key	Required	Description
arch	yes	The Java system property `os.arch`
os	yes	The Java system property `os.name`.
locale	yes	Required locales. Several locales can be specified, separated with spaces.
platform-version-id	See below	Platform version of requested JRE (only used for the `j2se` element)
known-platforms	yes	Platform versions of the JREs that are already locally available, i.e., that do not require an additional download of a JRE. This is a version string.

The JNLP Client must examine the HTTP response status code and MIME type to determine if the result was successful. The valid responses are described in Section 6.1.2.

The *known-platforms* field is a version string that contains the platform versions of the JREs that the JNLP Client can use to run an installer, e.g, "1.2 1.3". This allows the Web server to make sure that a potential installer for the extension can be run on the client system. Product versions, such as "1.2.2", are not appropriate for this field.

In a request, either the version-id or the platform-version-id must be specified. Both cannot be specified at the same time. The platform-version-id is only applicable when an extension that describes a JRE is requested. An example of this is described below.

The version string used in the request is not necessarily exact, e.g., "2.3+". The Web server must specify the exact version-id (product version) of the extension/JRE that is returned in the response by setting the HTTP header field: `x-java-jnlp-version-id`. The exact version returned must be one that matches the requested version string.

For example, given the following element:

```
<extension
   href="http://www.mysite.com/servlet/ext/coolaudio.jnlp"
   version="2.3.0 2.3.1"/>
```

The HTTP request would look like this (given that a Java 2 SE 1.2 JRE is available):

```
http://www.mysite.com/servlet/ext/coolaudio.jnlp?arch=x86&os=Windows+95&
locale=en_US&version-id=2.3.0+2.3.1&known-platforms=1.2
```

The above request is based on the version-based protocol. That request could also include the `current-version-id` element if an extension-descriptor was already downloaded.

Given the following element:

```
<extension href="http://www.mysite.com/ext/coolaudio.jnlp"/>
```

The HTTP request would be using the basic download protocol, and look like this:

```
http://www.mysite.com/ext/coolaudio.jnlp
```

A JRE can be downloaded using the extension protocol. The j2se element contains two attributes, version and href, which guide the installation process. For example:

```
<j2se version="1.3"
   href="http://www.jrevendor.com/servlet/jreinstaller"/>
```

The version and href parameters serve the same purpose as in the extension element. Hence, the HTTP GET request will look like:

```
http://www.jrevendor.com/servlet/jreinstaller?arch=x86&os=Windows+95&loc
ale=en_US&version-id=1.3&known-platforms=1.2
```

If an href attribute is not specified (which should be the most common case), then a platform version of the Java 2 platform is requested. If no JRE that implements that particular version is available on the client machine, the extension download protocol can be used to download an implementation that does. The *platform-version-id* argument will be used in the request, instead of the *version-id* argument. For

example:

```
<j2se version="1.3"/>
```

The JNLP Client is responsible for knowing a URL from which it can download an extension that implements that particular version. For example, the HTTP GET request could look like:

```
http://jsp.java.sun.com/servlet/javawsExtensionInstaller?arch=x86&os=Win
dows+95&locale=en_US&platform-version-id=1.3&known-platforms=1.2
```

6.5 Cache Management

A JNLP Client may cache the downloaded resources locally and is encouraged to do so. Resources are cached differently depending on whether a version-id is associated with it or not. The following caching rules apply to nativelib, jar, icon resources, and extension descriptors. How an application descriptor is downloaded and cached is described in Section 6.6.

6.5.1 Caching a resource without a version

An entry downloaded using the basic download protocol must be located in the cache based on the URL. The time stamp obtained from the HTTP GET request in the `Last-Modified` header field of the reply should be stored along with the downloaded resource. The time stamp is used to determine if the copy on the server is newer.

The JNLP Client cannot assume that the HTTP GET request will return the same JAR file for each request. The JNLP Client must periodically check the Web server to see if an updated version is available. This check is recommended to be performed before an application is launched, but the exact algorithm used by a JNLP Client depends on the particular implementation. For example, if a JNLP Client is offline, the check is not required to be performed.

The above caching rules also apply to extension descriptors downloaded using the extension download protocol where the version attribute is not specified.

6.5.2 Caching a resource with a version

An entry downloaded with the version-based download protocol must be cached using the URL and the (exact) version-id from the HTTP response as a key. When a lookup is performed given a URL and a version string, any resource cached using the given URL and with a version-id that matches the version string can be returned.

The JNLP Client can assume that the reply contents from an version-based request with an exact version-id is always the same. Thus, no time-stamp information needs to be stored. The resource is uniquely identified with the URL and version-id. If a given resource (URL/version-id pair) is already found in the cache, then the Web server does not need to be contacted to check for a newer version. Thus, using the version-based download protocol can provide better performance at startup, since potentially fewer connections need to be made back to the Web server.

The above caching rules also apply to extension descriptors downloaded using the extension download protocol where the version attribute is specified.

6.5.3 Managing the Cache

The JNLP Client is responsible for managing the cache of downloaded resources. The JNLP Client must make sure that the following invariant is maintained:

- Resources belonging to a particular application are never removed from the cache while the application is running.

This rule makes sure that the application developer can make assumptions about resources being in the cache while the application is running. In particular, all resources that are eagerly downloaded are going to be available locally in the cache during the entire program execution.

The exact policy and algorithms used to manage the cache are implementation-dependent. A reasonable policy might be to first clear out resources that are marked *lazy* before the ones marked *eager*.

The JNLP Client can also manage extensions in any way it sees fit. They can be uninstalled at any given point or be kept around permanently. If the extension uninstaller is invoked, then another request for the extension will require it to be downloaded again and the extension installer to be rerun.

6.6 Downloading and Caching of Application Descriptors

Application descriptors, i.e., JNLP files, are handled specially, since they are not necessarily downloaded by the JNLP Client itself. Often, they will be downloaded by a Web browser or by other means.

The href attribute in jnlp element is used to specify the location of the JNLP file itself. For example:

```
<jnlp href="http://www.mysite.com/app/App.jnlp">
```

If the href attribute is specified, then the JNLP file can be cached, and it also allows a JNLP Client to query the Web server for a newer version of the JNLP file, i.e., for the application. A JNLP Client can use this feature to, e.g., inform the user of updates to already-cached applications, or to automatically update applications during non-peak hours. In order to do this, a JNLP Client could keep track of the JNLP files it has downloaded, and then periodically query the Web server for new versions.

A JNLP file must be downloaded with an HTTP GET request to the specified URL. The JNLP Client must use the `Last-Modified` header field returned by the Web Server to determine if a newer JNLP file is present on the Web server.

7 JNLP API

A JNLP Client must provide a set of additional services to the launched application through the `javax.jnlp` package. These services let an application interact with the surrounding environment in a secure and platform-independent way. The JNLP API is available to all applications whether or not a particular application is trusted. Appendix D contains a list of all methods and classes with complete signatures. For a detailed description of the JNLP API, consult the JNLP API Reference, v1.0.

A service is structured as an object implementing a specific JNLP service interface. The following services are defined:

- `BasicService`, which provides a service similar to the AppletContext. This service must always be provided.
- `DownloadService`, which allows an application to interact with the JNLP Client to check if application resources are available locally, and to request them to be downloaded. This service must always be provided.
- `FileOpenService`, which allows applications running in the untrusted environment to import files from the local disk. This service is optional.
- `FileSaveService`, which allows applications running in the untrusted environment to export files to the local disk. This service is optional.
- `ClipboardService`, which allows applications running in the untrusted environment access to the clipboard. This service is optional.
- `PrintService`, which allows applications running in the untrusted environment access to printing. This service is optional.
- `PersistenceService`, which allows applications running in the untrusted environment access to store state locally on the client. This service is optional.
- `ExtensionInstallerService`, which provides an interface for extension installers to communicate with the JNLP Client. This service is required.

A service object is found using the static `lookup` method on the `ServiceManager` class. This method will, given a `String` representing the service name, return an object implementing the given service. The `lookup` method must be idempotent, i.e., returning the same object for each request for the same service. If a service is not available, the `UnavailableServiceException` must be thrown. The `getServiceNames` methods returns the names of all supported services.

The recommended name for a service is the fully qualified name of the interface, e.g., `javax.jnlp.BasicService`.

7.1 The BasicService Service

The `javax.jnlp.BasicService` service provides a set of methods for querying and interacting with the environment similar to what the `AppletContext` provides for a Java Applet.

The `getCodebase` method returns the codebase for the application. This will typically be the URL specified in the `codebase` attribute in the `jnlp` element. However, if the JNLP file does not specify this attribute, then the codebase is defined to be the URL of the JAR file containing the class with the `main` method.

The `isOffline` method returns true if the application is running without access to the network. An application can use this method to adjust its behavior to work properly in an offline environment. The method provides a hint from the JNLP Client. The network might be unavailable, even though the JNLP Client indicated that it was, and vice-versa.

The `showDocument` method displays the given URL in a Web browser. This may be the default browser on the platform, or it may be chosen by the JNLP Client some other way. This method returns false if the request failed, or the operation is not supported.

Some platforms might not support a browser, or the JNLP Client might not be configured to use a browser. The `isWebBrowserSupported` method will return true if a Web browser is supported, otherwise false.

7.2 The DownloadService Service

The `javax.jnlp.DownloadService` service allows an application to control how its own resources are cached.

The service allows an application to determine which of its resources are currently cached, to force resources to be cached, and to remove resources from the cache.

The following three query methods return `true` if a given resource, a given part, or a given part of a given extension is currently cached, respectively. The methods must always return `false` for resources that do not belong to the current application, i.e., are not mentioned in the JNLP file for the application.

- `isResourceCached`
- `isPartCached`
- `isExtensionPartCached`

The following three methods instruct the JNLP Client to download a given resource, a given part, or a given part of a given extension, respectively. The methods block until the download is completed or an error occurs. The methods must always fail for resources that do not belong to the current application, i.e., are not mentioned in the JNLP file for the application.

- `loadResource`
- `loadPart`
- `loadExtensionPart`

The above methods take an `DownloadServiceListener` object as argument that can track the progress of the download. A default implementation of a listener can be obtained by the `getDefaultProgressWindow` method.

The following three methods instruct the JNLP Client to remove a given resource, a given part, or a given part of a given extension from the cache, respectively. The remove request is a hint to the JNLP Client that the given resource is no longer needed. The methods must do nothing if a resource not belonging to the given application is requested to be removed.

- `removeResource`
- `removePart`
- `removeExtensionPart`

7.3 The FileOpenService Service

The `javax.jnlp.FileOpenService` service provides methods for importing files from the local disk, even for applications that are running in the untrusted execution environment.

This interface is designed to provide the same level of disk access to potentially untrusted Web-deployed applications that a Web developer has when using HTML. HTML forms support the inclusion of files by displaying a file open dialog.

A file open dialog can be displayed to the user with the following two methods:

- `openFileDialog`
- `openMultiFileDialog`

The methods allow selection of exactly one file or multiple files, respectively.

The contents of a file are returned in a `FileContents` object. A `FileContents` encapsulates the name of the selected file and provides metered access to the contents. The contents can be accessed using input streams, output streams, or random access. The JNLP Client might enforce size limits on the amount of data that can be written.

The `FileContents` only knows the name of the selected file excluding the path. Thus, the open dialog cannot be used to obtain information about the user's directory structure.

The JNLP Client can render the open file dialog in any way it sees fit. In particular, it could show an additional security dialog or warning message to the user before showing the dialog.

The methods will return `null` if the user chose to cancel the operation. An `IOException` will be thrown if the operation failed for some other reason than the user did not select a file.

On JNLP Clients running on disk-less systems, or systems that do not wish to implement these features, the `ServiceManager` must throw an `UnavailableServiceException` when this service is looked up.

7.4 The FileSaveService Service

The `javax.jnlp.FileSaveService` service provides methods for exporting files to the local disk, even for applications that are running in the untrusted execution environment.

This interface is designed to provide the same level of disk access to potentially untrusted Web-deployed Java applications, that a Web browser provides for contents that it is displaying. Most Web browsers provide a *Save As...* dialog as part of their user interface.

A file save dialog can be displayed to the user by invoking the `saveFileDialog` or the `saveAsFileDialog` methods.

The JNLP Client can render the save file dialog in any way it sees fit. In particular, it could show an additional security dialog or warning message to the user before an action is committed.

The methods return a `FileContents` object representing the file that was saved, or return `null` if the user chose to cancel the operation. An `IOException` will be thrown if the operation failed for some other reason than the user decided not to save the file.

On JNLP Clients running on disk-less systems, or systems that do not wish to implement this service, the `ServiceManager` must throw an `UnavailableServiceException` when this service is looked up.

7.5 The ClipboardService

The `javax.jnlp.ClipboardService` service provides methods for accessing the shared system-wide clipboard, even for applications that are running in the untrusted execution environment.

A JNLP Client implementing this service should warn the user of the potential security risk of letting an untrusted application access potentially confidential information stored in the clipboard, or overwriting contents stored in the clipboard.

The interface consists of two methods:

- `setContents`
- `getContents`

The two methods are analogues to the methods on `java.awt.datatransfer.Clipboard`, except that the JNLP API does not support owner notification or retrieving the name of the contents.

7.6 The PrintService Service

The `javax.jnlp.PrintService` service provides methods for accessing to printing even for applications that are running in the untrusted execution environment.

This service is designed to provide (somewhat) similar access to printing as an HTML-based application has through the browser. Using this service, an application can submit a print job to the JNLP Client. The JNLP Client can then show this request to the user, and if accepted queue the request to the printer.

The service provides a `print` method that can either take a `Pageable` object or a `Printable` object. The method will return `true` if the printing succeeded, otherwise it will return `false`.

The interface also provides a set of methods to get the current page format:

- `getDefaultPage`
- `showPageFormatDialog`

7.7 The PersistenceService Service

The `javax.jnlp.PersistenceService` service provides methods for storing data locally on the client system, even for applications that are running in the untrusted execution environment.

The service is designed to be (somewhat) similar to that which the cookie mechanism provides to HTML-based applications. Cookies allow a small amount of data to be stored locally on the client system. That data can be securely managed by the browser and can only be retrieved by HTML pages which originate from the same URL as the page that stored the data.

Each entry in the persistent data store is stored on the local system, indexed by a URL, i.e, using a URL as a key. This provides a similar hierarchical structure as a traditional file system. An application is only allowed to access data stored with a URL (i.e., key) that is based on its codebase. The URL must follow the directory structure of the codebase for a particular application. For example, given the codebase, `http://www.mysite.com/apps/App1/`, the application would be allowed to access data at the associated URLs:

- `http://www.mysite.com/apps/App1/`
- `http://www.mysite.com/apps/`
- `http://www.mysite.com/`

This scheme allows sharing of data between different applications from the same host. For example, if another application is located at `http://www.mysite.com/apps/App2/`, then they can share data between them in the `http://www.mysite.com/` and `http://www.mysite.com/apps/` directories. Any data that App1 wants to keep private from App2 can be stored at `http://www.mysite.com/apps/App1`.

The following methods are used to create, access, and delete entries. These methods all take a URL as an argument.

- `create`
- `get`
- `delete`

The `get` method returns the contents of the entry as a `FileContents` object. The `FileContents` interface provides metered access to the underlying data, i.e., ensures that the application does not write more data to disk than it is allowed. The `create` method takes an `long` argument that specifies that maximum size of the given entry. If this maximum size limit is exceeded when performing a write operation, an `IOException` must be thrown. The application can use the `setMaxSize` method on a `FileContents` object to request additional space.

A JNLP Client should keep track the amount of storage that a given application uses. The total amount of storage is the sum of the maximum sizes of all the entries that the application has access to. The default limit of the amount of total storage available to an application is JNLP Client specific, but should typically not be less than 128KB.

The `delete` method removes an entry from the persistence cache.

The `getNames` method returns the names of all entries in a given directory.

Data stored using this mechanism is intended to be a local copy of data stored on a remote server. Using a local cache can improve performance, as well as make it possible to run applications offline. The individual entries can be tagged as being either i) *cached*, meaning that the server has an up-to-date copy, ii) *dirty*, meaning that the server does not have an up-to-date copy, or iii) *temporary*, meaning that this file can always be recreated. The following two methods support tags:

- `setTag`
- `getTag`

The tag information can be used by a JNLP Client when cleaning out the persistent cache. Entries that are tagged as temporary should be removed first, followed by entries that are tagged as cached, and then finally the dirty ones should be removed. The JNLP Client should always warn the user that unsaved data might be lost when removing an entry marked as dirty.

7.8 The ExtensionInstallerService Service

The `javax.jnlp.ExtensionInstallerService` service provides methods for an extension installer to manipulate the progress screen during a download and install, as well as methods for informing the JNLP Client where to find the resources it installed.

The `ExtensionInstallerService` is only available when the JNLP Client is running an extension installer. Otherwise, the `ServiceManager` should throw an `UnavailableServiceException` when this service is looked up.

An extension installer is a Java Technology-based application that is responsible for installing platform-dependent code and settings for an extension.

The extension installer provides three kinds of services to the installer:

- The ability to manipulate a progress window provided by the JNLP Client. The calls can be ignored if the JNLP Client does not implement a progress window.
- Query the JNLP Client about the preferred location for the installation and previous installations.
- Update the JNLP Client with information about the native parts of an extension. This will either be a list of native libraries that need to be linked into an application that uses the extension, or if the extension installs a JRE, how the JRE is to be launched.

The manipulation of the progress window is done using the following methods:

- `setStatus`
- `setHeading`
- `updateProgress`
- `hideProgressBar`
- `hideStatusWindow`

The inclusion of a progress window API should make it possible to write installers that look good on a wide variety of platforms. The JNLP Client will be responsible for the look and feel of the window. The progress window is assumed to already be showing when the extension installer is launched, e.g., it has just been used by the JNLP Client itself to show that the extension is being installed.

The `getInstallPath` provides a preferred location where the extension should be installed. A JNLP Client will typically create a directory under its own tree and return the location of that directory. The installer can then install all required files without creating a conflict with other installed extensions.

The `getExtensionVersion` and `getExtensionLocation` methods return the exact version-id and location of the extension that is being installed. This information would typically already be known by the installer, but is provided so that it is possible to write generic installers that work with multiple versions.

The `setJREInfo` and `setNativeLibraryInfo` methods are used to update the JNLP Client with information about the native code an installer might install. Only one of them can be called by a particular extension. The `setJREInfo` is used by an extension that installs a JRE. It must be called with the path to the executable that launches the JVM. The `setNativeLibraryInfo` method is used to instruct the JNLP Client to include the given directory in the search path for native libraries when the `System.loadLibrary` method is used.

The `installSucceeded` method must be called when the installer finished successfully. The `installFailed` method must be called if the installation failed. In case of a failed install, the installer is responsible for providing an error message to the user. After either of the methods are called, the installer should quit, and the JNLP Client should regain control and continue with its operation, e.g., continue launching the application that forced the extension to be downloaded and installed, or abort the launch if the the installation failed. An installer should not call `System.exit`.

The following method can be used by an installer to figure out the location of already installed JREs:

- `getInstalledJRE`

This allow an extension to potentially update a particular JRE by installing JAR files into, e.g., the `lib/ext` directory.

The normal sequence of events for an extension installer is:

Step 1: Get the ExtensionInstallerService from the ServiceManager. The installer can then use the `getInstallPath` to find out the preferred installation directory.

Step 2: Update status, heading, and progress as the install progresses (`setStatus`, `setHeading`, `updateProgress`, and `hideProgressBar`). If the installer uses its own progress window, the JNLP Client supplied one can be disposed using the `hideStatusWindow` method.

Step 3: If successful, inform the JNLP Client about the installed contents, by invoking either `setJREInfo` or `setNativeLibraryInfo`, as appropriate for the installer.

Step 4: Inform the JNLP Client about the completion of the installer. If successful, invoke `installSucceeded`. If not successful invoke `installFailed`. This will cause the JNLP Client to regain control, and continue with the launch sequence. An installer that must reboot the client system indicates that in the call of `installSucceeded`.

8 Future Directions

Many excellent suggestions for additions to this specification have been made by contributors from both our partners and internal reviewers. The desire for a timely specification constrains the amount of work that can be done for any particular revision of the specification, and so some of these suggestions cannot be incorporated in this version of the specification. However, by including these items as future directions, we indicate that we will be considering them for inclusion into a future revision of the specification.

The following items are under consideration:

- Fine-grained security directives
- URL-independent naming for extensions and JAR resources to enhance scalability and reliability of downloads.

Please note that the inclusion of an item on this list is not a commitment for inclusion into a future revision of this specification, only that the item is under serious consideration and may be included into a future revision.

A VERSION IDS AND VERSION STRINGS

This section is simply a formal encoding of common conventions for dot-notations. The formal syntax is to ensure predictable behavior of the download protocols.

This section describes the formal syntax of the version-id's and version strings used in this specification. A version-id is an exact version that is associated with a resource, such as a JAR file. A version string is a key that can match one or more version-id's.

The version-id used in this specification must conform to the following syntax:

```
version-id ::= string ( separator string ) *
string     ::= char ( char ) *
char       ::= Any ASCII character except a space, a separator or a
               modifier
separator ::= "." | "-" | "_"
```

A version string is a list of version-id's separated with spaces. Each version-id can be postfixed with a '+' to indicate a greater-than-or-equal match, a "*" to indicated a prefix match, or have no postfix to indicate an exact match. The syntax of version-strings is:

```
version-string ::= element ( " " element) *
element        ::= version-id modifier?
modifier       ::= "+" | "*"
```

A version-id can be described as a tuple of values. A version-id string is broken in parts for each separator ('.', '-', or '_'). For example, "1.3.0-rc2-w" becomes (1,3,0,rc2,w), and "1.2.2-001" becomes (1,2,2,001).

Each element in a tuple is treated as either a numeric or alphanumeric. Two elements are compared numerically if they can both be parsed as Java `ints`, otherwise they are compared lexicographically according to the ASCII value[16] of the individual characters.

Before two version-id's are compared the two tuples are *normalized*. This means that the shortest tuple is padded with 0 (zero element) entries at the end. Two normalized tuples are always of the same length. For example, comparing (1, 3) and (1, 3, 1), will result in comparing (1, 3, 0) and (1, 3, 1).

A.1 ORDERING

The version-id's are ordered by the natural ordering of dot-notations.

A normalized version-id tuple can be written as (*Head Tail)*, where *Head* is the first element in the tuple, and *Tail* is the rest[17].

Given two version-id's, (HA TA) and (HB TB), then (HA TA) is greater than (HB TB) if and only if:

- HA is greater than HB, or
- HA is equal to HB, TA and TB are not empty, and TA is greater than TB recursively

16 The specification restricts the version-id's to only contain ASCII characters due to the well-defined ordering of ASCII characters based on the ASCII value.

17 This is treating the tuple as e.g. a Lisp list.

In other words, A is greater than B if, when represented as normalized tuples, there exists some element of A which is greater than the corresponding element of B, and all earlier elements of A are the same as in B.

For example, "1.2.2" is greater than "1.2", and less than "1.3" (i.e., in effect, comparing "1.2.2", "1.2.0", and "1.3.0")

A.2 Exact Match

Two normalized version-id's, (HA TA) and (HB TB), match exactly if and only if:

- HA is equal to HB and
- TA and TB are both empty, or TA matches TB exactly.

In other words, A is an exact match of B if, when represented as normalized tuples, the elements of A are the same as the elements of B.

For example, given the above definition "1.2.2-004" will be an exact match for "1.2.2.4", and "1.3" is an exact match of "1.3.0".

A.3 Prefix Match

Given two version-id's, (HA TA) and (HB TB), then first (HB TB) is padded with 0 (zero element) entries at the end so it is at least the same length as the (HA TA) tuple.

(HA TA) is a prefix match of (HB TB) if and only if:

- HA is equal to HB, and
    - TA is empty, or
    - TA is a prefix match of TB

In other words, A is a prefix match of B if, when represented as tuples, the elements of A are the same as the first elements of B. The padding ensures that B has at least as many elements as A.

For example, given the above definition "1.2.1" will be a prefix match to "1.2.1-004", but not to "1.2.0" or "1.2.10". The padding step ensures that "1.2.0.0" is a prefix of "1.2". Note that prefix matching and ordering are distinct: "1.3" is greater than "1.2", and less than "1.4", but not a prefix of either.

B JARDiff Format

This format describes how to apply incremental updates to a JAR file. An incremental update can be applied to an already-downloaded JAR file to yield an updated version. Downloading an incremental update to an existing version can significantly reduce download time compared to downloading the new JAR file, if the existing and new JAR files have most parts in common.

For example, given two JAR files: `from.jar` and `to.jar`, then a JARDiff can be computed that describes the changes that need to be applied to `from.jar` to yield `to.jar`.

B.1 MIME Type and Default File Extension

The default MIME type and extension that should be associated with a JARDiff file are shown in the following table.

Default MIME Type	Default Extension
`application/x-java-archive-diff`	`.jardiff`

B.2 Contents

The JARDiff file is itself a JAR file.

In the following, it is assumed that the original JAR file is named `from.jar`, and the updated JAR file is named `to.jar`. A JARDiff between `from.jar` and `to.jar` contains the following:

- The set of entries that exist in `to.jar` but do not exist in `from.jar`, except for entries that have just been renamed.
- The set of entries that exist in `from.jar`, but are modified in `to.jar`.
- An index file, `META-INF/INDEX.JD,` that describes the contents of the `to.jar` file, and how it relates to the `from.jar` file. The `INDEX.JD` filename should be generated in upper case, but should be recognized in any case. This file is always required.

Thus, a JARDiff file contains complete copies of each new or changed file. It does not provide a way to incrementally update individual files within a JAR file.

B.3 The Index File

The index file describes what entries from from.jar to include in the target file. The file contains commands of the form:

Command	Meaning
`version <id>`	Version of the JARDiff protocol.
`remove <entry>`	Do not include the <entry> from `from.jar` in `to.jar`
`move <from> <to>`	Include the entry <from> from `from.jar` in `to.jar` as <to>

The backslash (\) is used as an escape character. A backslash is represented as two slashes (\\), and a space as '\ ', i.e., a slash followed by a space. The backslash is used only as an escape character; it does not define any special characters. For example, \t represents the character t, and \i represents the character i.

The index file must be UTF-8 encoded.

The commands are used as follows:

- The `version` command must always be the first entry in a index file. The current version is 1.0.
- The `remove` command means that the given file from `from.jar` should not be included in the target file.
- The `move` command means that the given file from `from.jar` should be included in the target file as the given name.

For each entry in the `to.jar` file there can either be a `remove` command in the index file, one or more `move` commands, or no entry at all. A `move` and `remove` command for the same file is invalid.

A file that does not appear in any move or remove command, and which *does not* appear in the JARDiff file, is copied from from.jar to to.jar as-is. Also, a file that does not appear in any move or remove command, which *does* appear in the JARDiff file, is copied from the JARDiff file to to.jar as-is. These two rules reduce the size of the index file.

B.4 Applying a JARDiff

The following pseudo-code shows how to apply a JARDiff:

```
Let old-names = List of entries in old.jar

// Add new and/or updated entries. This also takes
// care of implicit removes
for each x in JARDiff file except META-INF/INDEX.JD
   add the contents of x from JARDIFF to target JAR as x
   remove x from old-names
end
// Iterate through index file
for each cmd in META-INF/INDEX.JD do
   if cmd is 'remove x' then
       remove x from old-names
   else if cmd is 'move x y' then
       add the content of x from old.jar to target JAR as y
       remove x from old-names
   end
end
// Do all implicit moves
for each x in old-names
   add the content of x from old.jar to target JAR as x
end
```

A JARDiff file that will cause the same filename to be added to the target file twice is invalid. Thus, the add command must fail if it the same file is added twice, and an error should be signaled.

B.5 Signing and JARDiff files

JARDiff files themselves are not signed. Instead, they can contain the signing information for the target file, i.e., the manifest, signature instructions, and digital signature. Thus, the target JAR file is signed if it can be verified using the standard procedure for a signed JAR file.

B.6 Example

The following shows an example of a JARDiff file.

Assume that the JAR file, `app.jar`, contains version 1.0 of an application:

```
com/mysite/app/Main.class
com/mysite/app/Window1.class
com/mysite/app/QuickHack.class
com/mysite/app/stuff.properties
```

Later on, version 1.1 of the application is released. The new `app.jar` contains the following entries:

```
com/mysite/app/Main.class
com/mysite/app/Window1.class
com/mysite/app/Window2.class
com/mysite/app/app.properties
```

An inspection of the differences between `app.jar` version 1.0 and version 1.1 yields the following differences:

- `Main.class` has been updated with support for a new application window.
- `stuff.properties` has been renamed to `app.properties`.
- `Window2.class` has been added in version 1.1.
- `QuickHack.class` does not exist in version 1.1
- `Window1.class` is unchanged.

The difference between app.jar version 1.0 and 1.1 can be expressed by a JARDiff file containing the following entries (all from version 1.1):

```
META-INF/INDEX.JD
com/mysite/app/Main.class
com/mysite/app/Window2.class
```

Thus, the JARDiff file contains all the new or modified files in version 1.1 compared to 1.0. The INDEX.JD file will list the following requests:

```
version 1.0
remove com/mysite/app/QuickHack.class
move com/mysite/app/stuff.properties com/mysite/app/app.properties
```

C JNLP File Document Type Definition

The following contains an annotated XML Document Type Definition (DTD) for the the JNLP file.

C.1 DOCTYPE

```
<!DOCTYPE jnlp-descriptor PUBLIC "-//Sun Microsystems, Inc//DTD JNLP
Descriptor 1.0//EN" "http://java.sun.com/products/j2se/dtds/
jnlp_1_0.dtd">
```

C.2 DTD

```
<!--
The root element for the JNLP file.
-->

<!ELEMENT jnlp (information+, security?, resources*, (application-desc |
applet-desc | component-desc | installer-desc))>

<!--
The spec attribute of the jnlp element specifies what versions of the
JNLP specification a particular JNLP file works with. The default value
is "1.0+".
-->

<!ATTLIST jnlp spec CDATA #IMPLIED>

<!--
The version attribute of the jnlp element specifies the version of the
application being launched, as well as the version of the JNLP file
itself.
-->

<!ATTLIST jnlp version CDATA #IMPLIED>

<!--
The codebase attribute of the jnlp element specifies the codebase for
the application. This is also used as the base URL for all relative URLs
in href attributes.
-->

<!ATTLIST jnlp codebase CDATA #IMPLIED>

<!--
The href attribute of the jnlp element contains the location of the JNLP
file as a URL.
-->

<!ATTLIST jnlp href CDATA #IMPLIED>
```

```
<!--
The information element contains various descriptive information about
the application being launched.
-->

<!ELEMENT information (title?, vendor?, homepage?, description*, icon*,
offline-allowed?)>

<!--
The locale attribute of the information element specifies the locale for
which this information element should be used.
-->

<!ATTLIST information locale CDATA #IMPLIED>

<!--
The title element contains the name of the application.
-->

<!ELEMENT title (#PCDATA)>

<!--
The vendor element contains the name of the vendor.
-->

<!ELEMENT vendor (#PCDATA)>

<!--
The homepage element contains a href to the homepage for the
application.
-->

<!ELEMENT homepage EMPTY>

<!--
The href attribute of the homepage element specifies the URL for the
homepage.
-->

<!ATTLIST homepage href CDATA #REQUIRED>


<!--
The description element contains a description of the application.
-->

<!ELEMENT description (#PCDATA)>
```

```
<!--
The kind attribute for the description element indicates the use of a
description element. The values are: i) one-line, for a one-line
description, ii) short, for a one paragraph description, and iii)
tooltip, for a tool-tip description. Longer descriptions should be put
on a separate web page and referred to using the homepage element.
-->

<!ATTLIST description kind (one-line | short | tooltip) #IMPLIED>

<!--
The icon element describes an image for an application.
-->

<!ELEMENT icon EMPTY>

<!--
The href attribute of an icon contains a URL to a location on the web
containing an image file for an icon. The file must be in either JPEG or
GIF format.
-->

<!ATTLIST icon href CDATA #REQUIRED>

<!--
The version attribute of an icon contains a string describing the
version of the image that is requested.
-->

<!ATTLIST icon version CDATA #IMPLIED>

<!--
The width attribute of the icon element describes the width of the icon
in pixels.
-->

<!ATTLIST icon width CDATA #IMPLIED>

<!--
The height attribute of the icon element describes the height of the
icon in pixels.
-->

<!ATTLIST icon height CDATA #IMPLIED>

<!--
The kind attribute of the icon element describes the use of the icon.
-->

<!ATTLIST icon kind (default | selected | disabled | rollover)
"default">
```

```
<!--
The depth attribute of the icon element describes the color depth of the
image in bits-per-pixel. Common values will be 8, 16, or 24.
-->

<!ATTLIST icon depth CDATA #IMPLIED>


<!--
The size attribute of an icon element indicates the size of an icon file
in bytes.
-->

<!ATTLIST icon size CDATA #IMPLIED>

<!--
The offline-allowed element indicates if the application can be launched
offline. Default value (i.e., if the element is not specified) is
online.
-->

<!ELEMENT offline-allowed EMPTY>

<!--
The security element describes the security requirements of the
application.
-->

<!ELEMENT security (all-permissions?, j2ee-application-client-
permissions?)>

<!--
The all-permissions element indicates that the application needs full
access the the local system and network.
-->

<!ELEMENT all-permissions EMPTY>

<!--
The j2ee-application-client-permissions element indicates that the
application needs the set of permissions defined for a J2EE application
client.
-->

<!ELEMENT j2ee-application-client-permissions EMPTY>

<!--
The resources element contains an ordered set of resources that
constitutes an application.
-->

<!ELEMENT resources (j2se | jar | nativelib | extension | property |
package)*>
```

```
<!--
The os attribute of the resources element specifies for which operating
system this element should be considered.
-->

<!ATTLIST resources os CDATA #IMPLIED>

<!--
The arch attribute of the resources element specifies for what platform
this element should be considered.
-->

<!ATTLIST resources arch CDATA #IMPLIED>

<!--
The locale attribute of the resources element specifies for which
locales this element should be considered.
-->

<!ATTLIST resources locale CDATA #IMPLIED>

<!--
The j2se element describes a supported JRE version and an optional
resources element to be used by the particular JRE.
-->

<!ELEMENT j2se (resources*)>

<!--
The version attribute of the j2se element describes the versions of the
JRE that this application is supported on.
-->

<!ATTLIST j2se version CDATA #REQUIRED>

<!--
The href attribute of the j2se element specifies the location where the
JRE should be downloaded from.
-->

<!ATTLIST j2se href CDATA #IMPLIED>

<!--
The initial-heap-size attribute of the j2se element specifies the
initial size of the object heap.
-->

<!ATTLIST j2se initial-heap-size CDATA #IMPLIED>
```

```
<!--
The max-heap-size attribute of the j2se element specifies the preferred
maximum size of the object heap.
-->

<!ATTLIST j2se max-heap-size CDATA #IMPLIED>

<!--
The jar element describes a jar file resource.
-->

<!ELEMENT jar EMPTY>

<!--
The href attribute of the jar element contains the location of a jar
file as a URL.
-->

<!ATTLIST jar href CDATA #REQUIRED>

<!--
The version attribute of a jar element describes the version of a
particular JAR file that is requested.
-->

<!ATTLIST jar version CDATA #IMPLIED>

<!--
The main attribute of a jar element indicates whether this element
contains the main class.
-->

<!ATTLIST jar main (true|false) "false">

<!--
The download attribute of a jar element indicates if this element must
be downloaded before an application is launched (eager), or not (lazy).
-->

<!ATTLIST jar download (eager | lazy) "eager">

<!--
The size attribute of a jar element indicates the size of a JAR file in
bytes.
-->

<!ATTLIST jar size CDATA #IMPLIED>
```

```
<!--
The part attribute of a jar element describes the name of the group it
belongs too.
-->

<!ATTLIST jar part CDATA #IMPLIED>

<!--
The nativelib element describes a resource containing native files.
-->

<!ELEMENT nativelib EMPTY>

<!--
The href attribute of a nativelib element contains the location of a
nativelib file as a URL.
-->

<!ATTLIST nativelib href CDATA #REQUIRED>

<!--
The version attribute of a nativelib element describes the version of a
particular nativelib file that is requested.
-->

<!ATTLIST nativelib version CDATA #IMPLIED>

<!--
The download attribute of a nativelib element indicates if this element
must be downloaded before an application is launched (eager), or not
(lazy).
-->

<!ATTLIST nativelib download (eager | lazy) "eager">

<!--
The size attribute of a nativelib element indicates the size of a
nativelib file in bytes.
-->

<!ATTLIST nativelib size CDATA #IMPLIED>

<!--
The part attribute of a nativelib element describes the name of the part
it belongs to.
-->

<!ATTLIST nativelib part CDATA #IMPLIED>

<!--
The extension element describes an extension that is required in order
to run the application.
-->

<!ELEMENT extension (ext-download*)>
```

```
<!--
The version attribute of an extension element specifies the version of
the extension requested.
-->

<!ATTLIST extension version CDATA #IMPLIED>

<!--
The name attribute of an extension element specifies the name of the
extension.
-->

<!ATTLIST extension name CDATA #IMPLIED>

<!--
The href attribute of an  extension element specifies the location of
the extension.
-->

<!ATTLIST extension href CDATA #REQUIRED>

<!--
The ext-download element defines how parts of the extension are
downloaded.
-->

<!ELEMENT ext-download EMPTY>

<!--
The ext-part attribute of an ext-download element describes the name of
a part in the extension.
-->

<!ATTLIST ext-download ext-part CDATA #REQUIRED>

<!--
The download attribute of an ext-download element describes if the
resource may be lazily downloaded.
-->

<!ATTLIST ext-download download (lazy|eager) "eager">

<!--
The part attribute of an ext-download element describes the name of the
part it belongs to in the current JNLP file.
-->

<!ATTLIST ext-download part CDATA #IMPLIED>

<!--
The property element describes a name/value pair that is available to
the launched application as a system property.
-->

<!ELEMENT property EMPTY>
```

```
<!--
The name attribute of the property element describes the name of a
system property.
-->

<!ATTLIST property name CDATA #REQUIRED>

<!--
The value element describes the value of a system property.
-->

<!ATTLIST property value CDATA #REQUIRED>

<!--
The package element defines a relationship between a Java package or
class name and a part.
-->

<!ELEMENT package EMPTY>

<!--
The name attribute of the package element describes the name of a
package or class.
-->

<!ATTLIST package name CDATA #REQUIRED>

<!--
The part attribute of the package element describes the part that
contains the specified package or class.
-->

<!ATTLIST package part CDATA #REQUIRED>

<!--
The recursive attribute of the package element indicates if all sub-
packages of this particular package is also included.
-->

<!ATTLIST package recursive (true|false) "false">

<!--
The application-desc element describes how to launch a Java-based
application. It contains information about the main class and arguments.
-->

<!ELEMENT application-desc (argument*)>

<!--
The main-class attribute of the application-desc element describes the
main class of an application.
-->

<!ATTLIST application-desc main-class CDATA #IMPLIED>
```

```
<!--
The argument elements describe the ordered set of arguments to an
application. These arguments will be passed into the main method of the
application's main class.
-->

<!ELEMENT argument (#PCDATA)>

<!--
The applet-desc element describes how to launch a Java Technology-based
Applet. It contains information about, e.g., the main class, size, and
parameters.
-->

<!ELEMENT applet-desc (param*)>

<!--
The documentbase attribute of the applet-desc element describes the
documentbase for the applet as a URL.
-->

<!ATTLIST applet-desc documentbase CDATA #IMPLIED>

<!--
The main-class attribute of the applet-desc element describes the name
of the main Applet class.
-->

<!ATTLIST applet-desc main-class CDATA #REQUIRED>

<!--
The name attribute of the applet-desc element describes the name of the
Applet.
-->

<!ATTLIST applet-desc name CDATA #REQUIRED>

<!--
The width attribute of the applet-desc element describes the width of
the Applet in pixels.
-->

<!ATTLIST applet-desc width CDATA #REQUIRED>

<!--
The height attribute of the applet-desc element describes the height of
the Applet in pixels.
-->

<!ATTLIST applet-desc height CDATA #REQUIRED>

<!--
The param element describes a parameter to an Applet.
-->

<!ELEMENT param EMPTY>
```

```
<!--
The name attribute of the param element describes the name of a
parameter.
-->

<!ATTLIST param name CDATA #REQUIRED>

<!--
The value attribute of the param element describes the value of a
parameter.
-->

<!ATTLIST param value CDATA #REQUIRED>
<!--
The component-desc element specifies a component extension.
-->

<!ELEMENT component-desc EMPTY>

<!--
The installer-desc element specifies an installer extension.
-->

<!ELEMENT installer-desc EMPTY>

<!--
The main-class attribute of the installer-desc element describes the
main class for the installer/uninstaller.
-->

<!ATTLIST installer-desc main-class CDATA #IMPLIED>
```

D Application Programming Interface

This is a listing of the interfaces, classes, and exceptions that compose the JNLP API. For detailed descriptions of these members and their methods, please see the JNLP API Reference, v1.0.

D.1 JNLP API Package Summary

The table below summarizes the classes and interfaces comprising the JNLP API.

Package javax.jnlp	Service Name	Required
BasicService	javax.jnlp.BasicService	yes
DownloadService	javax.jnlp.DownloadService	yes
FileOpenService	javax.jnlp.FileOpenService	no
FileSaveService	javax.jnlp.FileSaveService	no
ClipboardService	javax.jnlp.ClipboardService	no
PrintService	javax.jnlp.PrintService	no
PersistenceService	javax.jnlp.PersistenceService	no
ExtensionInstallerService	javax.jnlp.ExtensionInstallerService	yes
UnavailableServiceException	<*not a service*>	n/a
DownloadServiceListener	<*not a service*>	n/a
FileContents	<*not a service*>	n/a
JNLPRandomAccessFile	<*not a service*>	n/a
ServiceManager	<*not a service*>	n/a
ServiceManagerStub	<*not a service*>	n/a

D.2 ServiceManager

```
public final class ServiceManager

static public Object lookup(String name) throws
  UnavailableServiceException;
static public String[] getServiceNames();
static public void setServiceManagerStub(ServiceManagerStub stub);
```

D.3 ServiceManagerStub

```
public interface ServiceManagerStub

public Object lookup(String name) throws
  UnavailableServiceException;
public String[] getServiceNames();
```

D.4 BasicService

```
import java.net.URL;
```

```
public interface BasicService

public URL getCodeBase();
public boolean isOffline();
public boolean showDocument(URL url);
public boolean isWebBrowserSupported();
```

D.5 DownloadService

```
import java.net.URL;

public interface DownloadService

public boolean isResourceCached(URL ref, String version);
public boolean isPartCached(String part);
public boolean isPartCached(String[] parts);
public boolean isExtensionPartCached(URL ref, String version, String
   part);
public boolean isExtensionPartCached(URL ref, String version,
   String[] parts);
public void loadResource(URL ref, String version,
   DownloadServiceListener listener) throws IOException;
public void loadPart(String part, DownloadServiceListener listener)
   throws IOException;
public void loadPart(String[] parts, DownloadServiceListener listener)
   throws IOException;
public void loadExtensionPart(URL ref, String version, String part,
   DownloadServiceListener listener) throws IOException;
public void loadExtensionPart(URL ref, String version, String[] parts,
   DownloadServiceListener listener) throws IOException;
public void removeResource(URL ref, String version) throws IOException;
public void removePart(String part) throws IOException;
public void removePart(String[] parts) throws IOException;
public void removeExtensionPart(URl ref, String version, String part)
   throws IOException;
public void removeExtensionPart(URl ref, String version, String parts)
throw IOException;
public DownloadServiceListener getDefaultProgressWindow();
```

D.6 FileOpenService

```
import java.io.IOException;

public interface FileOpenService

public FileContents openFileDialog(String pathHint, String[]
  exts) throws IOException;
public FileContents[] openMultiFileDialog(String pathHint,
  String[] exts) throws IOException;
```

D.7 FileSaveService

```
import java.io.IOException;
import java.io.InputStream;

public interface FileSaveService
```

```
public FileContents saveFileDialog(String pathHint, String[]
   extensions, InputStream stream, String name) throws IOException;
public FileContents saveAsFileDialog(String pathHint, String[]
   extensions, FileContents contents) throws IOException;
```

D.8 ClipboardService

```
import java.awt.datatransfer.Transferable;

public interface ClipboardService

public Transferable getContents();
public void setContents(Transferable contents);
```

D.9 PrintService

```
import java.awt.print.Pageable;
import java.awt.print.Printable;
import java.awt.print.PageFormat;

public interface PrintingService

public PageFormat getDefaultPage();
public PageFormat showPageFormatDialog(PageFormat page);
public boolean print(Pageable document);
public boolean print(Printable painter);
```

D.10 PersistenceService

```
import java.io.InputStream;
import java.io.OutputStream;
import java.io.RandomAccessFile;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.URL;
import java.net.MalformedURLException

public interface PersistenceService

public static int final CACHED = 0;
public static int final TEMPORARY = 1;
public static int final DIRTY = 2;

public long create(URL url, long maxSize)
   throws MalformedURLException, IOException;
public FileContents get(URL url)
   throws MalformedURLException, FileNotFoundException, IOException;
public void delete(URL url)
   throws MalformedURLException, IOException;
public String[] getNames(URL url)
   throws MalformedURLException, IOException;
public int getTag(URL url)
   throws MalformedURLException, IOException;
public void setTag(URL url, int tag)
```

```
   throws MalformedURLException, IOException;
```

D.11 ExtensionInstallerService

```
public interface ExtensionInstallerService

public String getInstallPath();
public String getExtensionVersion();
public URL getExtensionLocation();
public void hideProgressBar();
public void hideStatusWindow();
public void setHeading(String heading);
public void setStatus(String status);
public void updateProgress(float value);
public void installFailed();
public void installSucceeded(boolean needsReboot);
public void setJREInfo(String platformVersion, String jrePath);
public void setNativeLibraryDirectory(String path);
public String getInstalledJRE(URL location, String productVersion);
```

D.12 FileContents

```
import java.io.IOException;

public interface FileContents

public String getName() throws IOException;
public boolean canRead() throws IOException;
public boolean canWrite() throws IOException;
public long getLength() throws IOException;
public long getMaxLength() throws IOException;
public long setMaxLength(long maxlength) throws IOException;
public InputStream getInputStream() throws IOException;
public OutputStream getOutputStream(boolean overwrite) throws
IOException;
public JNLPRandomAccessFile getRandomAccessFile(String mode) throws
IOException;
```

D.13 JNLPRandomAccessFile

```
import java.io.IOException;

public interface JNLPRandomAccessFile[18]
extends java.io.DataInput, java.io.DataOutput

public void close() throws IOException;
public long length() throws IOException;
public long getFilePointer() throws IOException;
public int read() throws IOException;
public int read(byte [] b, int off, int len) throws IOException;
public int read(byte [] b) throws IOException;
public void readFully(byte [] b) throws IOException;
public void readFully(byte b[], int off, int len) throws IOException;
public int skipBytes(int n) throws IOException;
```

18 The `java.io.RandomAccessFile` is not used since most of its methods are final. Thus, it would be impossible to implement a JNLP Client that returns a subclass of `RandomAccssFile` that implements, e.g., metered access to the file system.

```
public boolean readBoolean() throws IOException;
public byte readByte() throws IOException;
public int readUnsignedByte() throws IOException;
public short readShort() throws IOException;
public int readUnsignedShort() throws IOException;
public char readChar() throws IOException;
public int readInt() throws IOException;
public long readLong() throws IOException;
public float readFloat() throws IOException;
public double readDouble() throws IOException;
public String readLine() throws IOException;
public String readUTF() throws IOException;
public void seek(long pos) throws IOException;
public void setLength(long newLength) throws IOException;
public void write(int b) throws IOException;
public void write(byte b[]) throws IOException;
public void write(byte b[], int off, int len) throws IOException;
public void writeBoolean(boolean v) throws IOException;
public void writeByte(int v) throws IOException;
public void writeShort(int v) throws IOException;
public void writeChar(int v) throws IOException;
public void writeInt(int v) throws IOException;
public void writeLong(long v) throws IOException;
public void writeFloat(float v) throws IOException;
public void writeDouble(double v) throws IOException;
public void writeBytes(String s) throws IOException;
public void writeChars(String s) throws IOException;
public void writeUTF(String str) throws IOException;
```

D.14 UnavailableServiceException

```
public class UnavailableServiceException extends Exception

UnavailableServiceException();
UnavailableServiceException(String msg);
```

D.15 DownloadServiceListener

```
public interface DownloadServiceListener

public void progress(java.net.URL url, java.lang.String version,
   long readSoFar, long total, int overallPercent);
public void validating(java.net.URL url, java.lang.String version,
   long entry, long total, int overallPercent);
public void upgradingArchive(java.net.URL url, java.lang.String version,
   int patchPercent, int overallPercent);
public void downloadFailed(java.net.URL url, java.lang.String version);
```