



# ***Guide to AllJoyn™ Development Using the Java SDK***

***HT80-BA002-1 Rev. J***

***February 5, 2013***

---

**Submit technical questions at:**

**<http://www.alljoyn.org/forums>**

The information contained in this document is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License; provided, that (i) any source code incorporated in this document is licensed under the Apache License version 2.0 **AND (ii) THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN ARE PROVIDED ON AN "AS-IS" BASIS WITHOUT WARRANTY OF ANY KIND.**

[Creative Commons Attribution-ShareAlike 3.0 Unported License](#)

AllJoyn is a trademark of Qualcomm Innovation Center, Inc. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

**Qualcomm Innovation Center, Inc.  
5775 Morehouse Drive  
San Diego, CA 92121  
U.S.A.**

# Contents

---

<b>1 Introduction.....</b>	<b>6</b>
1.1 Scope.....	6
1.2 Audience.....	6
1.3 Prerequisites.....	6
1.4 Revision history.....	6
1.5 Related documents.....	7
<b>2 Importing Existing Java Samples into Eclipse.....</b>	<b>8</b>
<b>3 Starting a New Android Project.....</b>	<b>9</b>
<b>4 Starting a New Project for Desktop Development.....</b>	<b>10</b>
<b>5 Writing Code.....</b>	<b>11</b>
5.1 Loading the native library.....	11
5.2 Creating an AllJoyn interface.....	11
5.3 Using @BusInterface annotation.....	12
5.3.1 Using @BusMethod annotation.....	12
5.3.2 Using @BusSignal annotation.....	13
5.3.3 Using @BusProperty annotation.....	13
5.4 More complex AllJoyn interface concepts.....	13
5.4.1 Making a secure bus interface using the @Secure annotation.....	13
5.4.2 Complex data types using the @Position annotation.....	14
5.4.3 Multiple return values.....	15
5.5 Connecting to the AllJoyn daemon.....	15
5.5.1 Connecting a service.....	15
5.5.2 Connecting a client.....	17
5.6 Discovery and sessions.....	19
5.6.1 Full user control.....	19
5.6.2 How discovery works.....	19
5.6.3 Sample code.....	20
5.7 Using AllJoyn methods.....	22
5.8 Using AllJoyn signals.....	26
5.8.1 Sessionless Signals.....	30

5.9 Using AllJoyn properties.....	31
5.10 Security.....	33
5.10.1 AllJoyn's authentication mechanisms.....	34
5.10.2 The AuthListener.....	34
5.10.3 Code for SRP key exchange.....	35
5.10.4 Code for SRP logon.....	39
5.10.5 Code for RSA key exchange.....	44
5.10.6 The default keystore listener.....	56
5.11 AllJoyn's debug log output.....	58
5.11.1 Set OS-specific logging option.....	58
5.11.2 Set Debug Level for modules.....	58
5.11.3 Set ER_DEBUG_* environment variable.....	59
5.11.4 Log output.....	60
<b>6 Android-Specific Considerations.....</b>	<b>61</b>
6.1 UI thread vs. AllJoyn bus thread.....	61
6.2 Using Handler Looper.....	61
6.3 Security.....	62
6.4 Peer Permission Check.....	62
6.5 Bundled Daemon (Deprecated).....	63
6.6 Using ICE with AllJoyn.....	64
6.7 Permissions to Add to your AllJoyn Android App.....	64
6.8 ProGuard.....	65
<b>7 Performance.....</b>	<b>68</b>
7.1 Header compression.....	68
7.2 Shutting off discovery.....	68
7.3 Signal emitters.....	68

## Figures

Figure 1: MSC 1 SRP key exchange.....	36
Figure 2: MSC 2 SRP logon option 1.....	39
Figure 3: MSC 3 SRP logon option 2.....	40
Figure 4: MSC 4 SRP logon option 3.....	40
Figure 5: MSC 5 SRP logon option 4.....	41

Figure 6: MSC 6 RSA Key Exchange option 1.....	45
Figure 7: MSC 7 RSA Key Exchange option 2.....	46
Figure 8: MSC 8 RSA Key Exchange option 3.....	47
Figure 9: MSC 9 RSA Key Exchange option 4.....	48

## Tables

Table 1: Revision history.....	6
Table 2: Related documents.....	7
Table 3: AllJoyn request name flags.....	17
Table 4: Results from a RequestName call.....	17
Table 5: Authentication mechanisms and AllJoyn identifying string.....	34
Table 6: OS-specific logging behavior.....	58

## Code Examples

Code 1. Load native library.....	11
Code 2. Example of an interface.....	11
Code 3. Specifying signature and replySignature.....	12
Code 4. Auto assignment of signatures.....	13
Code 5. Creating a secure interface.....	14
Code 6. Example data structure using @Position annotation.....	14
Code 7. Method that accepts multiple return values.....	15
Code 8. Connecting a service to the bus (no error checking).....	16
Code 9. Connecting a service (with error checking).....	16
Code 10. Connecting a client (with error checking).....	18
Code 11. Multiple interfaces.....	18
Code 12. Obtaining a well-known name/advertising the well-known name.....	20
Code 13. Find well-known name and join session.....	21
Code 14. Sample interface with AllJoyn methods.....	22
Code 15. Implementing the SampleInterface in the Service.....	22

Code 16. Connecting the bus using the SampleService BusObject.....	23
Code 17. Calling a remote method from the SampleInterface.....	24
Code 18. Calling the remote methods.....	26
Code 19. Example interface containing signals.....	27
Code 20. Create an empty class implementation.....	27
Code 21. Connect signals to the bus and emitting the signals (service).....	27
Code 22. Registering signal handlers (client).....	29
Code 23. Adding signal handlers (client).....	29
Code 24. Interface using properties.....	31
Code 25. Implementation of the PropertiesInterface (service).....	31
Code 26. Connecting the Properties interface to the bus (service).....	32
Code 27. Calling the properties interface and using the GetAll method for properties.....	32
Code 28. Registering the authentication listener.....	36
Code 29. SRP key exchange AuthListener (service).....	37
Code 30. SRP key exchange AuthListener (client).....	38
Code 31. Registering the authentication listener for SRP logon.....	42
Code 32. SRP Logon AuthListener (service).....	42
Code 33. SRP Logon AuthListener (client).....	43
Code 34. Registering the authentication listener.....	48
Code 35. RSA key exchange AuthListener (service and client).....	49
Code 36. RSA Key Exchange AuthListener (service and client).....	51
Code 37. Obtaining the X.509 certificate for verification.....	55
Code 38. Overriding the default file location when registering an AuthListener.....	57
Code 39. Registering AuthListeners with a shared keystore.....	57
Code 40. BusHandler class that extends the Handler class.....	61
Code 41. Starting the BusHandler class.....	62
Code 42. Adding the @AccessPermission annotation for an AllJoyn Java interface.....	63
Code 43. Sample proguard.cfg file.....	65
Code 44. Header compression.....	68
Code 45. Shutting off Discovery.....	68

# 1 Introduction

---

## 1.1 Scope

This document is a guide to using the AllJoyn application framework to write Java applications and includes a section on Android device development. This guide should be used with the sample programs in the AllJoyn SDK. This guide refers to code snippets extracted from the sample programs.

## 1.2 Audience

This guide will be helpful to application developers writing Java applications for the Android platform.

## 1.3 Prerequisites

- This document uses many terms and concepts that are described in the document “Introduction to AllJoyn.” Therefore, it is strongly recommended that you read that document before beginning this document.
- Before proceeding with development as described in this document, make sure that you have set up the development environment as described in the document “Setting Up the Build Environment (Microsoft Windows Platform)” or “Setting Up the Build Environment (Linux Platform).”
- Have at least one device running Android OS version 2.2 (Froyo) or greater and running a chip based on the ARM 5 (or greater) instruction set.

## 1.4 Revision history

[Table 1](#) provides the revision history for this document.

**Table 1: Revision history**

Version	Date	Description
A	January 21, 2011	Initial release
B	June 22, 2011	Engineering updates
C	July 27, 2011	Engineering updates to the security section
D	November 22, 2011	Formatting changes
E	December 2, 2011	Added sections on Peer Permissions Check and Bundled Daemon
F	December 14, 2012	Added section on ProGuard
G	January 27, 2012	Revised section on Bundled Daemon

Version	Date	Description
H	September 20, 2012	Updated information about Bundled Daemon. Added sections for using ICE and adding permissions.
I	n/a	Not used.
J	February 5, 2013	Added sections on Sessionless Signals and AllJoyn's debug log output. Updated sample proguard.cfg file.

## 1.5 Related documents

[Table 2](#) is a list of supplemental documents and downloads that relate to the information in this guide.

**Table 2: Related documents**

Location	Title
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>Introduction to AllJoyn</i>
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>AllJoyn Android Environment Setup Guide</i>
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>C++ API Reference Manual</i>
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>Java API Reference Manual</i>
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>AllJoyn SDK</i>
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>Source Code</i>
<a href="http://www.alljoyn.org">http://www.alljoyn.org</a>	<i>Configuring the Build Environment</i>

## 2 Importing Existing Java Samples into Eclipse

---

There are several samples provided as part of AllJoyn. The samples may be found in the AllJoyn SDK in the java/samples folder.

**Note** If you wish to start a new AllJoyn project instead, see [Starting a New Project for Desktop Development](#).

1. Start Eclipse.
2. Select **File > Import**.
3. Select **General > Existing Projects into Workspace**.
4. Click **Next**.
5. Click **Browse** near the Select root directory text box.
6. Navigate to the samples folder and find the simple service sample (<AllJoyn Dist>/java/samples/simple/service).
7. Click **OK**.
8. Click **Finish**.

The simple service sample should now be imported into your workspace. The project may be giving you a few errors.

To remove the project errors:

1. Set Class Compatibility to **5.0**.
2. Select the project in the Package Explorer window. Press **Alt+Enter** to bring up Properties for the simple service.
3. Select the **Java Compiler** option.
4. Under the Java Compiler options, check the **Enable project specific settings** option.
5. Set the **Compiler compliance level** to **1.5**.
6. Click **OK**.

Rebuild the auto-generated gen folder.

- Click **Project > Clean**.

Use the same process to import any sample of interest.



## 3 Starting a New Android Project

---

In Eclipse, create a new Android project. (Open a browser and navigate to <http://developer.android.com> for instructions on how to do this.)

1. Go to your workspace and find the folder that contains the new Android project.

The project folder should contain the following (it may contain more):

- res folder
  - src folder
  - AndroidManifest.xml file
2. Create a new folder with the name **libs**.
  3. Copy the file **<AllJoyn dist folder>/java/jar/alljoyn.jar** to the new libs folder.
  4. In the libs folder, create another folder with the name **armeabi**.
  5. Copy the **liballjoyn\_java.so** file from the AllJoyn distribution to the **armeabi** folder. The file is in the AllJoyn distribution in the folder **<AllJoyn dist folder>/java/libs/liballjoyn\_java.so**.

You should now have a project directory structure that is similar to this (only a partial view of a project).

```
+<project name>
|\src
||-<source code files>
|\libs
||\armeabi
|||-liballjoyn_java.so
||-alljoyn.jar
|\res
||-<project resources>
|-AndroidManifest.xml
```

**Note** It may be simpler to copy an already existing libs folder from one of the samples found in **<AllJoyn dist folder>/java/samples**.

6. In Eclipse, right-click on your project and select the **Properties** option.
7. Select the **Java Build Path**.
8. Click **Add JARs**.
9. Select the **alljoyn.jar** file from the **<project>/libs** folder.

**Important** Make sure the files are from the Android distribution of AllJoyn and not the Linux distribution. The files will have identical names, but the Linux distribution is not compiled for the Arm processor used on most Android phones; it is designed for the PC x86 architecture.

## 4 Starting a New Project for Desktop Development

---

1. Create a new Java Project.
2. In the step where you define the Java build settings, select the 'Libraries' tab.
3. Click **Add External JARs...**
4. Select **alljoyn.jar** from the <alljoyn\_dist>/java/jar directory.
5. Click the arrow next to the alljoyn.jar listed in your JARs, which brings up a list of Source attachment, Javadoc location, Native library location, and Access rules for alljoyn.jar.
6. Double click **Native library location**, which brings up a dialog.
7. Click the **External Folder...** button.
8. Select the directory that contains liballjoyn\_java.so <alljoyn\_dist>/java/lib.
9. Double-click **Javadoc location**.
10. Click **Browse....**
11. Select the directory that contains the java docs for alljoyn.jar <alljoyn\_dist>/java/docs.
12. Click **Finish**.

**Note** To add AllJoyn to an already existing project, go to the project properties, select the 'Java Build Path' properties, and follow the steps listed above.

## 5 Writing Code

---

### 5.1 Loading the native library

You must load the native system library into your code at runtime.

To do so, add the following lines to your code.

#### Code 1. Load native library

```
static {  
    System.loadLibrary("alljoyn_java");  
}
```

### 5.2 Creating an AllJoyn interface

AllJoyn enables inter-process communication through an object. The object is defined as a bus interface.

Each interface can contain:

- Methods
- Signals
- Properties

#### Code 2. Example of an interface

```
@BusInterface (name = "org.my.interface.name")  
public interface MyInterface {  
  
    @BusMethod  
    public String MyMethod(String inStr) throws BusException;  
  
    @BusSignal  
    public void MySignal(String inStr) throws BusException;  
  
    @BusProperty  
    public String GetMyProperty() throws BusException;  
  
    @BusProperty  
    public void SetMyProperty(String myProperty) throws BusException;  
  
}
```

## 5.3 Using @BusInterface annotation

The @BusInterface annotation tells the code that this interface is an AllJoyn interface.

All bus interfaces must have a name. If you do not assign a name, a default name is assigned.

The default interface name is **<your package name>.<Interface name>**.

Generally, you should choose your interface name and not use the default. Set the interface name by specifying the name annotation property.

**Interface naming rules** (from <http://dbus.freedesktop.org/doc/dbus-specification.html>).

- Interfaces have names with type STRING, meaning that they must be valid UTF-8 characters.
- There are also some additional restrictions that apply to interface names, specifically:
  - Interface names are composed of one or more elements separated by a period (“.”) character. All elements *must* contain at least one character.
  - Each element *must* only contain the ASCII characters [A-Z][a-z][0-9]\_ and *must not* begin with a digit.
  - Interface names *must* contain at least one “.” (period) character (and thus at least two elements).
  - Interface names *must not* begin with a “.” (period) character.
  - Interface names *must not* exceed the maximum name length of 255 characters.

### 5.3.1 Using @BusMethod annotation

The @BusMethod annotation tells the Java compiler that this is a bus method. AllJoyn methods work almost identical to a regular method in Java. The major difference is that the AllJoyn methods execute on a different process or device. The method can accept multiple arguments and reply with multiple arguments. AllJoyn methods can have multiple return values, but Java methods only allow a single return value. To receive multiple return values requires a container that can hold the multiple return values. [Multiple return values](#) explains how to do this.

The @BusMethod annotation has four properties: annotation, name, signature, and replySignature.

Under normal circumstances, the values for the annotation properties can be determined by AllJoyn. However, there are instances in which the signature must be specified; for example you need to send an unsigned integer in a method. Since Java does not have an unsigned integer type, this must be specified in the @BusMethod annotation.

#### Code 3. Specifying signature and replySignature

```
@BusMethod(signature="u", replySignature="u")
public int MyMethod3(int unsignedArg) throws BusException;
```

AllJoyn is very good at automatically assigning a signature for the arguments specified in the method or signal. If given the code shown in [Code 4](#), note that the signature and replySignature are not specified as part of the `@BusMethod` annotation. AllJoyn automatically assigns this method a signature “sdib” and a replySignature of “i.” The only time you must specify the signature is when it cannot be obtained from the interface itself.

#### Code 4. Auto assignment of signatures

```
@BusMethod
public int MyMethod4(String sArg,
                    double dArg,
                    int iArg,
                    boolean bArg) throws BusException;
```

The valid values for the signature and replySignature are any values that are valid according to <http://dbus.freedesktop.org/doc/dbus-specification.html>.

### 5.3.2 Using @BusSignal annotation

The `@BusSignal` annotation specifies that the following code is an AllJoyn signal. Unlike methods, signals have no replies. Because of this, signals always have a return type of void.

Like methods, signals can take multiple arguments.

A signal is seen only if a program has registered a signal handler for that signal.

See [Using AllJoyn signals](#) to register and use a signal handler.

### 5.3.3 Using @BusProperty annotation

The `@BusProperty` annotation specifies that the following code is an AllJoyn property. AllJoyn properties are exactly like AllJoyn methods except they are specialized for get/set commands of a single value.

## 5.4 More complex AllJoyn interface concepts

### 5.4.1 Making a secure bus interface using the @Secure annotation

Indicating that an interface must use a secured authentication mechanism is as simple as adding the `@Secure` annotation to the interface.

**Code 5. Creating a secure interface**

```
@BusInterface(name = "org.sample.SecureInterface")
@Secure
public interface SecureInterface {

    @BusMethod
    String myMethod(String inStr) throws BusException;
}
```

The `@Secure` annotation informs the bus that no method call or signal can be sent without authentication.

The server must authenticate that the client is trusted using one of the built-in authentication protocols. After authentication is completed, all messages and signals have the body of the message encrypted. Only the authenticated client can read the message. Only the body of the message is encrypted. The header is left unencrypted and any bus can read the header and use it to forward the message to its destination.

For more information on the three built-in authentication mechanisms, see [AllJoyn's authentication mechanisms](#).

## 5.4.2 Complex data types using the `@Position` annotation

Beyond simple data types, AllJoyn can handle complex data types such as arrays, maps, and structs. In the case of arrays and maps, the data type can be handled by the AllJoyn code with no special action. However, structs require additional annotation.

AllJoyn must know the order of all elements of a struct so that it can marshal and unmarshal (i.e., serialize) the message. This is where the `@Position` annotation is used.

Say that you have a structure that holds information about a photo; the code could look something like this:

**Code 6. Example data structure using `@Position` annotation**

```
public class ImageInfo{
    @Position(0)
    public String fileName;
    @Position(1)
    public int isoValue;
    @Position(2)
    public int dateTaken;
}
```

The `@Position` annotation numbering must start from “0” (zero) and count up in increments of one. Skipping a number, like going from `@Position(1)` to `@Position(3)` without having an `@Position(2)` anywhere in the code, is a logic error.

The signature for all structures is “r” or the list of all the values of the structure inside parentheses. For example, the signature for the `ImageInfo` class would be “(sii)” in [Code 6](#)

### 5.4.3 Multiple return values

If for some reason an interface with multiple return values is used, a container must be made to hold those values. If you have a method called “timer” that returns a `startTime` and an `endTime`, an interface could be designed to handle the two return values.

#### Code 7. Method that accepts multiple return values

```
@BusInterface (name = "org.my.timerInterface")
public interface MultipleReturnValuesInterface {

    public class Values {
        @Position(0) public int startTime;
        @Position(1) public int endTime;
    }

    @BusMethod(replySignature="ii")
    public Values timer() throws BusException;
}
```

**Note** The `replySignature` has to be specified, or AllJoyn assumes the `replySignature` is a struct with the `replySignature` “(ii)”.

Parentheses indicate a struct, which is a single return value, while no parentheses indicate two separate return values.

## 5.5 Connecting to the AllJoyn daemon

When connecting with the bus, a program may act as a service, a client, or both.

### 5.5.1 Connecting a service

Connecting a service consists of several steps.

1. Create a new `BusAttachment`.
2. Register a `BusObject` with a given absolute path using the `BusAttachment`.
3. Connect the `BusAttachment` to the bus.
4. Request a well-known name from the bus.

**Code 8. Connecting a service to the bus (no error checking)**

```

mBus = new BusAttachment("applicationName");
mBus.registerBusObject(this, "/servicepath");
mBus.connect();
int flags = 0; //no request name flags
mBus.requestName("com.my.well.known.name", flags);

```

This code does not have any error handling; thus, it is a poor programming example. It does show what the functions are doing. The functions `BusAttachment.registerBusObject(...)`, `BusAttachment.connect()`, and `DBusProxyObj.RequestName(...)` all return status that should be checked.

**Code 9. Connecting a service (with error checking)**

```

mBus = new BusAttachment("applicationName");
Status status = mBus.registerBusObject(this, "/servicepath");
if (Status.OK != status) {
    System.out.println("BusAttachment.registerBusObject() failed: " + status);

    System.exit(0);
    return;
}

status = mBus.connect();
if (Status.OK != status) {
    System.out.println("BusAttachment.connect() failed: " + status);
    System.exit(0);
    return;
}

int flags = 0; //no request name flags
status = mBus.requestName("com.my.well.known.name", flags);
if (status != Status.OK) {
    System.out.println("BusAttachment.requestName failed: " + status);
    System.exit(0);
    return;
}

```

**Note** This code would have to be in a class that implements the `BusObject`. If it did not implement the `BusObject`, the following line would not work:

```
mBus.registerBusObject(this, "/servicepath");
```

The flags value is a bit-mask that is used to tell the bus how to respond if the well-known name requested already exists on the bus. It also tells the bus how to act if the well-known name is already owned and another application requests the same well-known name.

The flag options are:

- `BusAttachment.ALLJOYN_REQUEST_NAME_ALLOW_REPLACEMENT`



- `BusAttachment.ALLJOYN_REQUEST_NAME_REPLACE_EXISTING`
- `BusAttachment.ALLJOYN_REQUEST_NAME_DO_NOT_QUEUE`

**Table 3: AllJoyn request name flags**

AllJoyn flag name	Value
<code>ALLJOYN_REQUEST_NAME_ALLOW_REPLACEMENT</code>	0x1
<code>ALLJOYN_REQUEST_NAME_REPLACE_EXISTING</code>	0x2
<code>ALLJOYN_REQUEST_NAME_DO_NOT_QUEUE</code>	0x4

Using the value of 0x0 (zero) for the AllJoyn flag is a valid option. It means do not use any requestName flags. When using no flags to request a well-known name from the bus, if the name already exists on the bus, your name request is added to the queue of other applications that have also requested that name.

The `BusAttachment.requestName` returns one of the following status values:

**Table 4: Results from a RequestName call**

DBusProxyObj.RequestName result	Value
OK	1
<code>DBUS_REQUEST_NAME_REPLY_IN_QUEUE</code>	2
<code>DBUS_REQUEST_NAME_REPLY_EXISTS</code>	3
<code>DBUS_REQUEST_NAME_REPLY_ALREADY_OWNER</code>	4
<code>BUS_UNEXPECTED_DISPOSITION</code>	
<code>BUS_NOT_CONNECTED</code>	

The valid flag combinations and RequestName results are all specified and described at <http://dbus.freedesktop.org/doc/dbus-specification.html>.

## 5.5.2 Connecting a client

To connect a client:

1. Create a `BusAttachment`.
2. Connect the `BusAttachment` to the bus.
3. Make a proxy object with the service name, absolute service path, and the interface of interest.
4. Make a proxy connection with the specified AllJoyn interface.
5. (Optional) Register signal handlers if the interface of interest has any signals or if you are interested in any of the signals that are part of the D-Bus or AllJoyn specification.

**Code 10. Connecting a client (with error checking)**

```

mBus = new BusAttachment("applicationName");

Status status = mBus.connect();
if (Status.OK != status) {
    System.out.println("BusAttachment.connect() failed:" + status);
    System.exit(0);
    return;
}

mProxyObj = mBus.getProxyBusObject("org.my.well.known.name",
    "/servicepath",
    BusAttachment.SESSION_ID_ANY,
    new Class[] { MyInterface.class });

mMyInterface = mProxyObj.getInterface(MyInterface.class);

status = mBus.registerSignalHandlers(this);
if (status != Status.OK) {
    System.out.println("BusAttachment.registerSignalHandlers() failed:" +
status);
    System.exit(0);
    return;
}

```

mMyInterface can now be used to call methods found on the service.

The given well-known name and path can have multiple interfaces. When using getProxyBusObject(), list all of the interfaces you are interested in using. There could have been multiple interfaces available at the object path “/servicepath.” If there was a second interface called MySecondInterface and you also wanted to make a proxy connection to that interface, then you need to list both interfaces when calling getProxyBusObject().

When calling the getInterface() method, you are selecting one of the possible interfaces out of the interfaces obtained from the getProxyBusObject() method call.

**Code 11. Multiple interfaces**

```

mProxyObj = mBus.getProxyBusObject("org.my.well.known.name",
    "/servicepath",
    BusAttachment.SESSION_ID_ANY,
    new Class[] { MyInterface.class, MySecondInterface.class });

mMyInterface = mProxyObj.getInterface(MyInterface.class);
mMySecondInterface = mProxyObj.getInterface(MySecondInterface.class);

```

## 5.6 Discovery and sessions

### 5.6.1 Full user control

What makes AllJoyn exciting is the ability to discover remote buses and interact with the interfaces on those remote buses as if they are local.

The AllJoyn daemon runs in the background. If an application asks the daemon to look for a remote interface, the daemon tries to locate the requested service. When the daemon is started, it is told which transport protocols to use (i.e., Bluetooth, TCP/IP). If any device can make a connection using any of the available transport protocols, its services can then be discovered and used by all of the devices that have been discovered in a dynamically-formed, peer-to-peer network.

Once the well-known name has been discovered, your application can join a started communication session using a user-selected session port.

**Note** Valid SessionPort values range from 1 to 0xFFFF.

### 5.6.2 How discovery works

For a service and client to discover each other, they must agree upon three things:

- A well-known bus name that will be advertised
  - The object path
  - The session port number that will be used
1. To advertise a service, start with the procedure described in [Connecting a service](#).
    - a. Create a new BusAttachment with RemoteMessage.Receive.
    - b. Register your BusObject; the local implementation of your interface with the BusAttachment.
    - c. Register a BusListener; a bus listener may not be needed, depending on your application.
    - d. Connect the BusAttachment with the AllJoyn daemon.
    - e. Bind the session port.

As part of binding the session port you also need to create a SessionPort Listener that is used to respond to JoinSession requests from the client.

- f. Request a well-known name from the bus.
  - g. Advertise the same well-known name.
2. To connect to the advertised service:
  - a. Create a new BusAttachment with RemoteMessage.Receive.
  - b. Register a BusListener; in this case it is required for the foundAdvertisedName signal.

- c. Find the Advertised well-known name.
- d. When the foundAdvertisedName signal is received, join the session.

**Important** Although you can advertise different names than your service's well-known name, it would result in the bus not being able to use your service. It is an error to advertise a different name than your service's own well-known name.

### 5.6.3 Sample code

#### Code 12. Obtaining a well-known name/advertising the well-known name

```

BusAttachment mBus;
mBus = new BusAttachment(getClass().getName(),
BusAttachment.RemoteMessage.Receive);

Status status = mBus.registerBusObject(myBusObject, "/MyService");
if (status != Status.OK) {
    System.exit(0);
    return;
}

status = mBus.connect();
if (status != Status.OK) {
    System.exit(0);
    return;
}

private static final short CONTACT_PORT=42;

Mutable.ShortValue contactPort = new Mutable.ShortValue(CONTACT_PORT);

SessionOpts sessionOpts = new SessionOpts();
sessionOpts.traffic = SessionOpts.TRAFFIC_MESSAGES;
sessionOpts.isMultipoint = false;
sessionOpts.proximity = SessionOpts.PROXIMITY_ANY;
sessionOpts.transports = SessionOpts.TRANSPORT_ANY;

status = mBus.bindSessionPort(contactPort, sessionOpts,
new SessionPortListener() {
@Override
public boolean acceptSessionJoiner(short sessionPort,
String joiner, SessionOpts sessionOpts) {
if (sessionPort == CONTACT_PORT) {
return true;
} else {
return false;
}
}
});

```

```

Int flags = 0; //do not use any request name flags
status = mBus.requestName("com.my.well.known.name", flags);
if (status != Status.OK) {
    System.exit(0);
    return;
}

/*
 * Important: the well-known name advertised should be identical
 * to the well-known name requested from the bus. Using a
 * different name is a logic error.
 */
status = mBus.advertiseName("com.my.well.known.name",
    SessionOpts.TRANSPORT_ANY);
if (status != Status.OK) {
    /*
     * If we are unable to advertise the name, release
     * the well-known name from the local bus.
     */
    mBus.releaseName("com.my.well.known.name");
    System.exit(0);
    return;
}

if (status != Status.OK) {
    System.exit(0);
    return;
}

```

### Code 13. Find well-known name and join session

```

BusAttachment mBus;
mBus = new BusAttachment(getClass().getName(),
    BusAttachment.RemoteMessage.Receive);
private static final short CONTACT_PORT=42;

mBus.registerBusListener(new BusListener() {
    @Override
    public void foundAdvertisedName(String name,
        short transport,
        String namePrefix) {
        short contactPort = CONTACT_PORT;
        SessionOpts sessionOpts = new SessionOpts();
        Mutable.IntegerValue sessionId = new Mutable.IntegerValue();

        Status status = mBus.joinSession((String) msg.obj,
            contactPort,

```

```

sessionId,
sessionOpts,
new SessionListener());}
});

Status status = mBus.connect();
if (status != Status.OK) {
    System.exit(0);
    return;
}

status = mBus.findAdvertisedName("com.my.well.known.name");
if (status != Status.OK) {
    System.exit(0);
    return;
}

```

## 5.7 Using AllJoyn methods

So far, we have talked about making an interface and we have talked about how to connect to the bus and discover a remote bus using sessions. This section shows how to write an AllJoyn bus method that can be instantiated from a remote process.

Suppose you have the following interface:

### Code 14. Sample interface with AllJoyn methods

```

@BusInterface (name = "org.sample.busMethods")
public interface SampleInterface {

    @BusMethod
    public String Ping(String str) throws BusException;

    @BusMethod
    public String Concatenate(String arg1, String arg2) throws BusException;

    @BusMethod
    public int Fibonacci(int arg1) throws BusException;
}

```

### Code 15. Implementing the SampleInterface in the Service

```

class SampleService implements SampleInterface, BusObject {
    public String Ping(String str){
        return str;
    }
}

```

```

    public String Concatenate(String arg1, String arg2){
        return arg1+arg2;
    }

    public int Fibonacci(int arg1){
        int a=0,b=1;

        for (int i=0;i<arg1;i++){
            a=a+b;
            b=a-b;
        }
        return a;
    }
}

```

Now that the SampleInterface has an implementation, you must use a SampleService object when you are registering the bus object.

#### Code 16. Connecting the bus using the SampleService BusObject

```

SampleService mySampleService;
BusAttachment mBus;
mBus = new BusAttachment(getClass().getName(),
BusAttachment.RemoteMessage.Receive);

/*
 * This line of code makes the local interface available to the bus
 */
Status status = mBus.registerBusObject(mySampleService, "/MyService");
if (status != Status.OK) {
    System.exit(0);
    return;
}

status = mBus.connect();
if (status != Status.OK) {
    System.exit(0);
    return;
}

private static final short CONTACT_PORT=42;

Mutable.ShortValue contactPort = new Mutable.ShortValue(CONTACT_PORT);

SessionOpts sessionOpts = new SessionOpts();
sessionOpts.traffic = SessionOpts.TRAFFIC_MESSAGES;
sessionOpts.isMultipoint = false;
sessionOpts.proximity = SessionOpts.PROXIMITY_ANY;
sessionOpts.transports = SessionOpts.TRANSPORT_ANY;

```

```

status = mBus.bindSessionPort(contactPort, sessionOpts,
new SessionPortListener() {
@Override
public boolean acceptSessionJoiner(short sessionPort,
String joiner,
SessionOpts sessionOpts) {
if (sessionPort == CONTACT_PORT) {
return true;
} else {
return false;
}
}
});
if (status != Status.OK) {
System.exit(0);
return;
}

Int flags = 0; //do not use any request name flags
status = mBus.requestName("com.my.well.known.name", flags);
if (status != Status.OK) {
    System.exit(0);
    return;
}

status = mBus.advertiseName("com.my.well.known.name", SessionOpts.TRANSPORT_ANY);
if (status != Status.OK) {
    mBus.releaseName("com.my.well.known.name");
    System.exit(0);
    return;
}
}

```

### Code 17. Calling a remote method from the SampleInterface

```

private BusAttachment mBus;
mBus = new BusAttachment(getClass().getName(),
BusAttachment.RemoteMessage.Receive);

Status status = mBus.connect();
if (Status.OK != status) {
    System.exit(0);
    return;
}

private ProxyBusObject mProxyObj;
private SampleInterface mSampleInterface;

private static final short CONTACT_PORT=42;

```



```
mBus.registerBusListener(new BusListener() {
    @Override
    public void foundAdvertisedName(String name,
        short transport,
        String namePrefix) {
        short contactPort = CONTACT_PORT;
        SessionOpts sessionOpts = new SessionOpts();
        sessionOpts.traffic = SessionOpts.TRAFFIC_MESSAGES;
        sessionOpts.isMultipoint = false;
        sessionOpts.proximity = SessionOpts.PROXIMITY_ANY;
        sessionOpts.transports = SessionOpts.TRANSPORT_ANY;

        Mutable.IntegerValue sessionId = new Mutable.IntegerValue();

        Status status = mBus.joinSession(name,
            contactPort,
            sessionId,
            sessionOpts,
            new SessionListener());

        mProxyObj = mBus.getProxyBusObject("com.my.well.known.name",
            "/MyService",
            sessionId.value,
            new Class[] { SampleInterface.class });
        mSampleInterface = mProxyObj.getInterface(SampleInterface.class);
    });

    Status status = mBus.connect();
    if (status != Status.OK) {
        System.exit(0);
        return;
    }

    status = mBus.findAdvertisedName("com.my.well.known.name");
    if (status != Status.OK) {
        System.exit(0);
        return;
    }
}
```

Now that you have a proxy connection to the remote service through `mSampleInterface` you can just call it like you would if the code were implemented locally.

**Code 18. Calling the remote methods**

```
try {
    // will return "Hello World"
    String strPing = mSampleInterface.Ping("Hello World");
    // will return "The Eagle has landed!"
    String strCat = mSampleInterface.Concatenate("The Eagle ", "has landed!");

    // will return 3
    int fib = mSampleInterface.Fibonacci(4);
} catch (BusException e) {
    //Handle exception here.
}
```

Not all methods need to return a value and not all methods need input arguments.

**Note** Remote methods cannot be called until we have obtained an interface object from the `ProxyBusObject` class. Trying to call a remote method before obtaining the interface could result in your application crashing. The `mSampleInterface` is a `null` object until we have connected it to the service using the `ProxyBusObject.getInterface()` method.

## 5.8 Using AllJoyn signals

Signals, unlike methods, never return a value. Client applications that are interested in receiving a signal must register for that signal with the bus. The client must implement a signal handler to respond to the signal for which it has registered.

**Code 19. Example interface containing signals**

```
@BusInterface (name = "org.sample.busSignals")
public interface SampleInterface {

    @BusSignal
    public void buttonClicked(int id) throws BusException;

    @BusSignal
    public void playerPosition(int x, int y, int z) throws BusException;
}
```

**Code 20. Create an empty class implementation**

```
class SignalInterface implements SampleInterface, BusObject {
    public void buttonClicked(int id) { /*No code needed here*/ }
    public void playerPosition(int x, int y, int z) { /*No code needed here*/ }
}
```

A `SignalEmitter` is required to emit a signal. To create a `SignalEmitter`, you must have a `BusObject`. The code above creates a `BusObject` that is aware of the bus signals found in the `SampleInterface`. Signals do not need to have any code added to transmit a signal, you just inform the bus that a signal with a given interface exists.

Once a `SignalEmitter` is created, an interface can be made to send the actual signals. No coding is needed to emit the signals beyond defining and using the interface.

**Code 21. Connect signals to the bus and emitting the signals (service)**

```
private static final short CONTACT_PORT=42;
static int sessionId;
static String joinerName;

boolean sessionEstablished = false;

BusAttachment mBus;
mBus = new BusAttachment("AppName", BusAttachment.RemoteMessage.Receive);

SignalInterface mySignalInterface = new SignalInterface();

Status status = mBus.registerBusObject(mySignalInterface, "/MyService/Path");
if (status != Status.OK) {
    System.exit(0);
    return;
}

status = mBus.connect();
if (status != Status.OK) {
```

```

        System.exit(0);
        return;
    }

    int flags = 0; //do not use any request name flags
    status = mBus.requestName("com.my.well.known.name", flags);
    if (status != Status.OK) {
        System.exit(0);
        return;
    }

    status = mBus.advertiseName("com.my.well.known.name", SessionOpts.TRANSPORT_ANY);
    if (status != Status.OK) {
        System.out.println("Status = " + status);
        mBus.releaseName("com.my.well.known.name");
        System.exit(0);
        return;
    }

    Mutable.ShortValue contactPort = new Mutable.ShortValue(CONTACT_PORT);

    SessionOpts sessionOpts = new SessionOpts();
    sessionOpts.traffic = SessionOpts.TRAFFIC_MESSAGES;
    sessionOpts.isMultipoint = false;
    sessionOpts.proximity = SessionOpts.PROXIMITY_ANY;
    sessionOpts.transports = SessionOpts.TRANSPORT_ANY;

    status = mBus.bindSessionPort(contactPort, sessionOpts,
        new SessionPortListener() {
            public boolean acceptSessionJoiner(short sessionPort, String joiner,
                                                SessionOpts sessionOpts) {

                if (sessionPort == CONTACT_PORT) {
                    return true;
                } else {
                    return false;
                }
            }
        });

    public void sessionJoined(short sessionPort, int id, String joiner) {
        sessionId = id;
        joinerName = joiner;
        sessionEstablished = true;
    }
});

try {
    while (!sessionEstablished) {
        Thread.sleep(10);
    }

    SignalEmitter emitter = new SignalEmitter(mySignalInterface, joinerName,

```

```

                                sessionId,
SignalEmitter.GlobalBroadcast.Off);
myInterface = emitter.getInterface(SampleInterface.class);
// Emitting signals
myInterface.buttonClicked(1);
myInterface.playerPosition(12, 1, -24);

} catch (InterruptedException ex) {
    System.out.println("Interrupted");
} catch (BusException ex) {
    System.out.println("Bus Exception: " + ex.toString());
}

```

### Code 22. Registering signal handlers (client)

```

status = mBus.registerSignalHandlers(this);
if (status != Status.OK) {
    System.exit(0);
    return;
}

```

When registering for signal handlers, you must implement a class that contains a method with the `@BusSignalHandler` annotation. In [Code 22](#), the signal handler would need to be in the same class. If needed, a different class can be used to handle the signals; just put that class as an argument for the `registerSignalHandlers` method.

### Code 23. Adding signal handlers (client)

```

@BusSignalHandler(iface="org.sample.busSignals", signal="buttonClicked")
public void buttonClicked(int id) {
    switch(id)
    case 0:
        startNewGame();
        break;
    case 1:
        continueGame();
        break;
    case 2:
        quitGame();
        break;
    default:
        break;
}

@BusSignalHandler(iface="org.sample.busSignals", signal="playerPosition")
public void playerPosition(int x, int y, int z) {

```

```
        updatePlayerPosition(x,y,z);  
    }
```

The important part of the `BusSignalHandler` is to get the `iface` name and the signal arguments correct. If these are not correct, it does not catch the emitted signal. The function name itself is not important; it could be changed from `public void buttonClicked(int id)` to `public void foo(int id)` and it would still catch the "buttonClicked" signal.

### 5.8.1 Sessionless Signals

A signal is a way to send one-way acknowledgement data from one peer to another or to multiple peers. However, when sending a signal we typically need to be a part of a session. There are many use cases where using a session to send a signal would be unnecessary overhead. For example, when you want to inform a peer, who is not in session with you, of a changed state of information. This works two ways, with a peer specifying whether he wants to receive such notifications and with the peer who is sending out this notification to anyone who subscribed for it.

The way to send sessionless signals is not much different than sending a regular signal. Just set the flags specifying that it is a sessionless signal and set the session id used to send the signal to 0 since we are not a part of a session when we send this out.

A sessionless signal does not need a session id - thus the name. The sample showing this simple feature is under `SDK/java/samples/sessionless`.

Along the lines of the above-mentioned sample, let us assume that a client peer is sending out sessionless signals. A service peer is the one that has subscribed for these signals/notifications.

Two things to know in the case of a sessionless signal is that:

1. It has a session id of 0.
2. It has the sessionless flag set to indicate that it is a sessionless signal.

This case can look like the following:

```
SignalEmitter emitter = new SignalEmitter(mySignalService, 0,  
Signal.GlobalBroadcast.Off);  
emitter.setSessionlessFlag(true);
```

The service side, which wants to receive these sessionless signals, calls an `addMatch()` function and passes in the rule for sessionless signals. This looks like:

```
Status status = mBus.addMatch("sessionless='t'");
```

That is all we need. The service side would have the signal handler for the signal sent by the client. No advertisements, no binding, or joining of sessions is necessary by any side.

## 5.9 Using AllJoyn properties

AllJoyn properties may be viewed as a specialized case of AllJoyn methods. An AllJoyn property must start with the word “get” or “set.” If it starts with the word “get,” it must return a single value and take no arguments. If it starts with the word “set,” it must take a single parameter and return void. The parameter can be a complex data type like an array or a structure. This code should closely resemble the code used for AllJoyn methods.

### Code 24. Interface using properties

```
@BusInterface (name = "org.samples.properties")
public interface PropertiesInterface {
    @BusProperty
    public int getTextSize() throws BusException;
    @BusProperty
    public void setTextSize(int size) throws BusException;
}
```

### Code 25. Implementation of the PropertiesInterface (service)

```
class AllJoynProperties implements PropertiesInterface, BusObject {
    private int mTextSize;

    public int getTextSize() {
        return mTextSize;
    }

    public void setTextSize(int size) {
        mTextSize = size;
    }
}
```

```

    }
}

```

### Code 26. Connecting the Properties interface to the bus (service)

```

private BusAttachment mBus;
mBus = new BusAttachment(getClass().getName(),
    BusAttachment.RemoteMessage.Receive);

AllJoynProperties mProperties = new AllJoynProperties();
Status status = mBus.registerBusObject(mProperties, "/testProperties");
if (status != Status.OK) {
    System.exit(0);
    return;
}

```

The D-Bus standard provides a way to get all of the properties from an interface. The code below shows how to use the interface implemented in [Code 25](#) as well as the GetAll method from the D-Bus standard to get the TextSize property.

**Note** The GetAll method can return multiple values into a Map. Each property can be accessed using the name of the property.

### Code 27. Calling the properties interface and using the GetAll method for properties

```

private BusAttachment mBus;
private ProxyBusObject mProxyObj;
private PropertiesInterface mPropertiesInterface;
private Properties mProperties;
private static short CONTACT_PORT = 42;

private boolean isJoined = false;

mBus = new BusAttachment(getClass().getName(),
    BusAttachment.RemoteMessage.Receive);

mBus.registerBusListener(new BusListener {
    public void foundAdvertisedName(String name, short transport, String
namePrefix) {
        short contactPort = CONTACT_PORT;
        SessionOpts sessionOpts = new SessionOpts();
        sessionOpts.traffic = SessionOpts.TRAFFIC_MESSAGES;
        sessionOpts.isMultipoint = false;
        sessionOpts.proximity = SessionOpts.PROXIMITY_ANY;
        sessionOpts.transports = SessionOpts.TRANSPORT_ANY;
    }
}

```



```

    Mutable.IntegerValue sessionId = new Mutable.IntegerValue();

    Status status = mBus.joinSession(name, contactPort, sessionId, sessionOpts,
new SessionListener());
    if (status != Status.OK) {
        System.exit(0);
    }
    mProxyObj = mBus.getProxyBusObject("com.my.well.known.name",
        "/testProperties",
        sessionId.value,
        new Class<?>[] { PropertiesInterface.class,
            Properties.class });

    mPropertiesInterface = mProxyObj.getInterface(PropertiesInterface.class);
    mProperties = mProxyObj.getInterface({Properties.class});
    isJoined = true;
}
});
Status status = mBus.connect();
if (Status.OK != status) {
    System.exit(0);
    return;
}

status = mBus.findAdvertisedName("com.my.well.known.name");
if (status != Status.OK) {
    System.exit(0);
}

while(!isJoined) {
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        System.out.println("Program interrupted");
    }
}

mPropertiesInterface.setTextSize(18);
int textSize = mPropertiesInterface.getTextSize();

Map<String, Variant> map = mProperties.GetAll("org.samples.properties");
textSize = map.get("TextSize").getObject(Integer.class);

```

## 5.10 Security

When an interface is marked as “secure”, it indicates that all data sent via a method call or signal must be encrypted. Only an authenticated client can use an interface that has

been marked as secure by using the `@Secure` annotation (see [Making a secure bus interface using the `@Secure` annotation](#)).

Marking an interface as a secure interface is as simple as adding the annotation `@Secure` when declaring the interface. See [Code 5](#).

### 5.10.1 AllJoyn's authentication mechanisms

When using a secure interface, an authentication listener must be created to handle authentication events. The authentication listener provides authentication credentials for the local application and, depending on the authentication mechanism, checks the credential of the remote application.

AllJoyn supports three types of authentication mechanisms:

- Secure Remote Password (SRP) anonymous key exchange
- Secure Remote Password (SRP) logon (username and password)
- RSA key exchange using X509 certificates

SRP key exchange uses a one-time authenticating password to set up an authentication key. This key is used for all subsequent interactions between the client and server.

SRP logon requires that the user give a username and password before the client and server are able to share an interface, unlike the SRP key exchange where the password exchange is a one-time event. SRP logon requires the username and password be supplied each time a session is established between the client and server.

RSA key exchange uses RSA public key cryptography to establish a shared-secret passphrase. The shared-secret passphrase is used to establish a key. This key is then used for all subsequent exchanges between the given client and server using the protocol.

Each type of security mechanism requires that an Authentication Listener be created that handles the specific type of security exchange needed. When creating the Authentication Listener, you specify which type of authentication mechanism you want to use. Each authentication mechanism is specified by a string.

**Table 5: Authentication mechanisms and AllJoyn identifying string**

Authentication mechanism	String identifying the authentication mechanism
SRP anonymous key exchange	ALLJOYN_SRP_KEYX
SRP logon	ALLJOYN_SRP_LOGON
RSA key exchange	ALLJOYN_RSA_KEYX

### 5.10.2 The AuthListener

The main difference between a secure application and a plain application, besides the `@Secure` annotation of the interfaces, is the **required** inclusion of an AuthListener.

An implementation of an AuthListener contains two callback methods:

- Requested
- Completed

Any time a method call comes in on a secure interface AllJoyn checks to see if it already knows how to decrypt the contents of the method call. If it cannot decrypt the method call, the requested(...) method call is called.

The requested method call contains some or all of the following information:

- The Authentication mechanism specified using one of the strings from [Table 5](#).
- PeerName—the well-known-name or unique name of the peer that initiated the Authentication request.
- A count of the number of authentication request attempts made. The count for the first Authentication request starts at 1.
- The user name of the user making the authentication request. A user name is not required for all forms of authentications and may be an empty string.
- The specific type of authentication information being requested. Depending on what type of authentication mechanism is being used, one or more of these requests may be made:
  - CertificateRequest (RSA key exchange)
  - LogonEntryRequest (SRP logon)
  - PasswordRequest (SRP key exchange, SRP logon, or RSA key exchange)
  - PrivateKeyRequest (RSA key exchange)
  - UserNameRequest (SRP logon)
  - VerifyRequest (RSA key exchange)

The requested(...) call back method may be called one or more times for each authentication request. For each request, the program is expected to obtain or generate the required information. The requested information can be obtained from any source available including requesting information from a user. If the requested information cannot be provided then returning false indicates failure.

Once the authentication has finished the completed(...) call-back method is called. This contains:

- The authentication mechanism that was just completed
- The peerName (well-known name or unique name)
- A boolean variable indicating if the authentication attempt completed successfully.

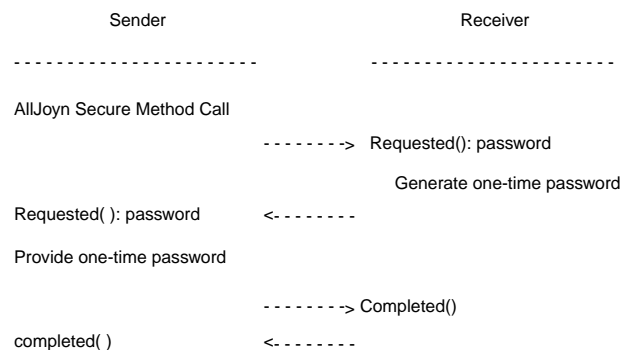
### 5.10.3 Code for SRP key exchange

SRP key exchange works similarly to pairing Bluetooth devices. The devices advertise that they want to talk with one another and when they discover one another a password is

entered on the phone. This one-time password proves that the phone knows about the Bluetooth device. Once this one-time pairing is completed, the Bluetooth device can connect with the phone at any time without repeating the pairing process.

[Code 28](#) sets up a client/server that uses SRP key exchange to set up a secure connection. The code shares many similarities with pairing a Bluetooth device.

- There is a one-time password exchange.
- Once the authentication is completed the client and server can communicate without asking for passwords to be entered.
- Even after the program is closed and restarted the authentication is still valid.



**Figure 1: MSC 1 SRP key exchange**

[Figure 1](#) shows the sequence in which the callback methods are instantiated the first time a secure method call is made using SRP key exchange. The sequence shown only happens the first time the sender tries to communicate with the receiver.

In [Code 28](#) the sender is the client and the receiver is the service.

Before calling `connect`, the `BusAttachment.registerAuthListener()` should be called. The code snippet below shows registering an `AuthListener` with the name `SrpKeyXListener`. The code for the `SrpKeyXListener` is shown in [Code 29](#) for the service and [Code 30](#) for the client.

#### Code 28. Registering the authentication listener

```

BusAttachment mBus;
mBus = new BusAttachment("myProgramName", BusAttachment.RemoteMessage.Receive);

SrpKeyXListener myAuthListener;

Status status = mBus.registerAuthListener("ALLJOYN_SRP_KEYX",
                                          myAuthListener);

if (status != Status.OK) {
    finish();
}
  
```

```

    return;
}

```

The rest of the steps for connecting and advertising the service have been covered in [Discovery and sessions](#) and are unchanged for a secure interface.

When registering the AuthListener we have identified what type of security the AuthListener is responsible for handling – in this case, "ALLJOYN\_SRP\_KEYX".

### Code 29. SRP key exchange AuthListener (service)

```

/*
 * This Secure Remote Password Key Exchange listener creates a random
 * 6-digit pin code for each authentication attempt.
 */
private static class SrpKeyXListener implements AuthListener {
    private Random mPasswordGenerator;
    private boolean mGeneratePassword;

    public SrpKeyXListener() {
        mPasswordGenerator = new Random();
        mGeneratePassword = true;
    }

    @Override
    public boolean requested(String mechanism, String peerName, int count,
                           String userName, AuthRequest[] requests) {
        if (mGeneratePassword) {
            mGeneratePassword = false;
            mPassword = String.format("%06d",
mPasswordGenerator.nextInt(1000000));
            System.out.println("One Time Password : " + mPassword);
        }
        for (AuthRequest request : requests) {
            if (request instanceof PasswordRequest) {
                ((PasswordRequest) request).setPassword(mPassword.toCharArray());
            }
        }
        return true;
    }

    @Override
    public void completed(String mechanism, String peerName, boolean
authenticated) {
        mGeneratePassword = true;
    }
}

```

```

    }
}

```

The first time that the `requested` callback is called, a six-digit random number is generated and set as the password. We check to see what `instanceof AuthRequest` we received. Since we are using SRP key exchange, we are only expecting password requests. We cast the base `AuthRequest` type to a `PasswordRequest` and call the `setPassword` method passing in a `CharArray` of the six-digit random number that was generated. The code shown does not limit the number of times an application can try to authenticate with this code. We can see the number of times the user has tried to authenticate by looking at the `count` parameter. This code only generates a new PIN code at the start of an authentication attempt; as long as `completed` has not been called, the same password is used.

### Code 30. SRP key exchange AuthListener (client)

```

/*
 * This Secure Remote Password Key eXchange listener will ask the user to
 * enter the one time password. This password must match the six-digit
 * string randomly generated by the service's SrpKeyXListener class.
 *
 * The following code limits the user to three authentication attempts.
 */
static class SrpKeyXListener implements AuthListener {
    public boolean requested(String authMechanism, String authPeer, int count,
        String userName,
                               AuthRequest[] requests) {
        if (count <= 3) {
            System.out.println("Please enter one time password:");
            Scanner in = new Scanner(System.in);
            mPassword = in.nextLine();
            for (AuthRequest request : requests) {
                if (request instanceof PasswordRequest) {
                    ((PasswordRequest)
request).setPassword(mPassword.toCharArray());
                }
            }
            return true;
        }
        return false;
    }

    public void completed(String authMechanism, String authPeer, boolean
authenticated) {
        if (!authenticated) {
            System.out.println("Authentication failed.");
        }
    }
}

```

```

    }
}

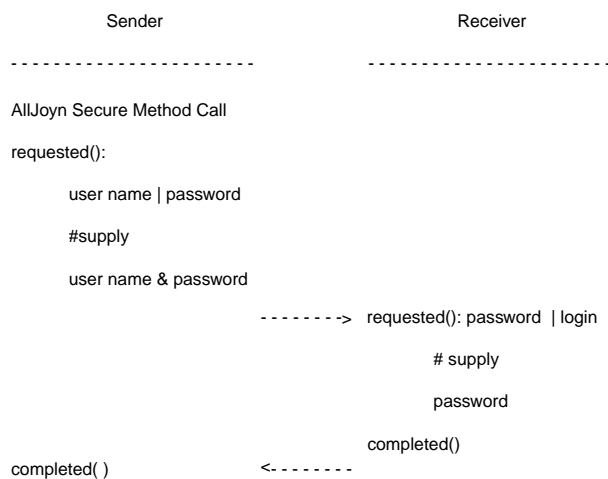
```

Each time that the `requested` callback is called the user is asked to provide the one-time password. The one-time password entered by the user must match the six-digit PIN code that was randomly generated by the service. The user is given three attempts to provide the proper PIN code. If the user is unable to provide the PIN code, the authentication attempt fails. Just like the code used in the service's `SrpKeyXListener`, we also check to see what `instanceof AuthRequest` we have received. Since we are using SRP key exchange, we are only expecting password requests. The base `AuthRequest` type is cast to a `PasswordRequest` type and the password is set using the `setPassword` method. Unlike the service, this code limits the number of attempts the user is given to produce the proper PIN code. After three failed attempts this returns false indicating failure to complete authentication.

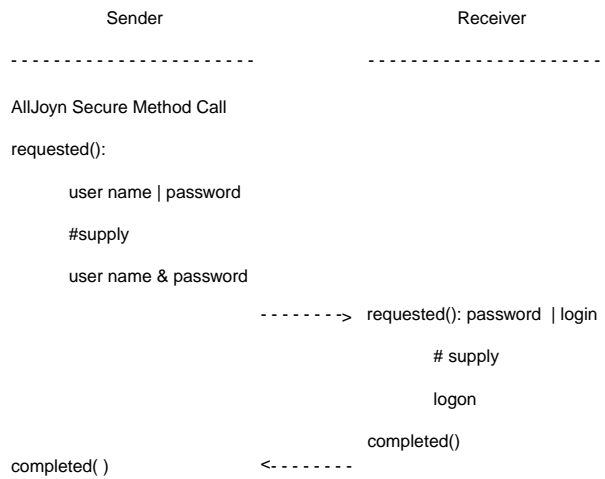
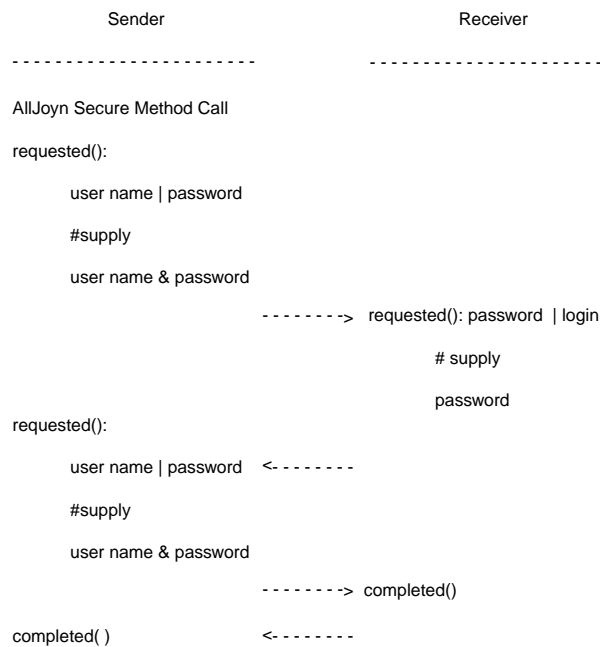
### 5.10.4 Code for SRP logon

SRP logon uses the familiar user name/password method for identity verification. This should be familiar to anyone who has used online shopping, email, or banking.

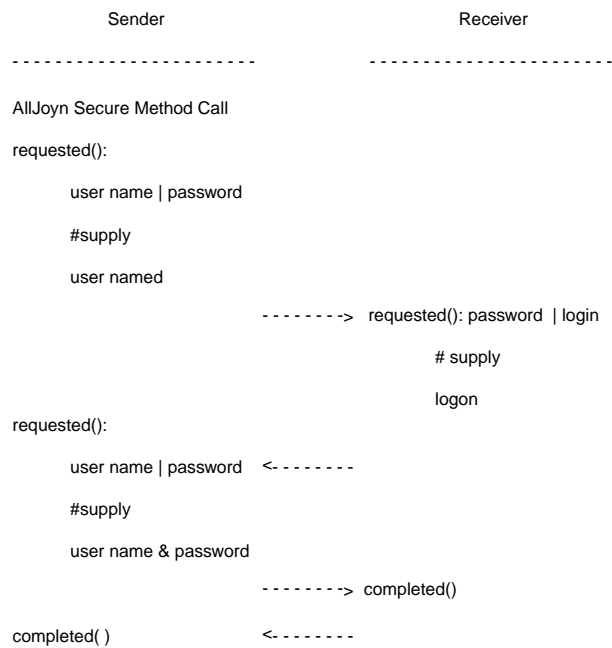
Unlike SRP key exchange and RSA key exchange, SRP logon requires the user to enter the user name/password every time a session is established. The verification only lasts for the length of the session. If the program is shut down and then restarted, the user is asked to re-authenticate.



**Figure 2: MSC 2 SRP logon option 1**

**Figure 3: MSC 3 SRP logon option 2****Figure 4: MSC 4 SRP logon option 3**





**Figure 5: MSC 5 SRP logon option 4**

With SRP logon, there are four possible sequence diagrams. When the sender is requested to provide a user name or password, they can provide just the user name or both the user name and password. The information is sent to the receiver and the receiver then uses the user name to look up the password or logon. If the password was also supplied and matches the receiver's password or the computer logon, the authentication completes. If only the user name was supplied, the receiver then requests the user name and password from the sender.

What is the logon? The logon is a string that is computed using the process described in RFC 5054. This ends up being a large string that takes the form of  $N:g:s:v$  where  $N$  is a large prime number,  $g$  is the generator,  $s$  is the salt and  $v$  is the computed verifier. The logon gives all the information needed to compute the verifier from the user name and password. This is a strongly encrypted string that can be used to verify a user name and password but cannot be used to obtain the user name and password. Because it is a secure way to store a user name and password, many databases do not actually store the password but instead store the logon as defined by RFC 5054. Computing the logon string is beyond the scope of this document. [Code 31](#) uses plain text user name and password.

When using a secure interface, an authentication listener must be created that handles authentication. The authentication listener decides if the connecting client is a trusted client and if they should be allowed to use the interface. The implementation of the `SrpLogonListener` is shown in [Code 31](#). See [Code 32](#) for the service (receiver) and [Code 33](#) for the client (sender).

**Code 31. Registering the authentication listener for SRP logon**

```

BusAttachment mBus;
mBus = new BusAttachment("myProgramName", BusAttachment.RemoteMessage.Receive);

SrpLogonListener myAuthListener;

Status status = mBus.registerAuthListener("ALLJOYN_SRP_LOGON", myAuthListener);
if (status != Status.OK) {
    finish();
    return;
}

```

The rest of the steps for connecting and advertising the service have been covered in the earlier sections and are unchanged for a secure interface.

**Code 32. SRP Logon AuthListener (service)**

```

class SrpLogonListener implements AuthListener {
    private Map<String, char[]> mUserNamePassword;

    /* Populate the user name and password table used by this listener. */
    public SrpLogonListener() {
        mUserNamePassword = new HashMap<String, char[]>();
        mUserNamePassword.put("user1", "password1".toCharArray());
        mUserNamePassword.put("user2", "password2".toCharArray());
    }

    /*
     * Given the user name, look up the password. Returning true without
     * setting the password tells the authentication engine to ask the peer
     * for the user name again.
     */
    public boolean requested(String mechanism,
                           int count,
                           String userName,
                           AuthRequest[] requests) {
        char[] password = mUserNamePassword.get(userName);
        if (password != null) {
            for (AuthRequest request : requests) {
                if (request instanceof PasswordRequest) {
                    ((PasswordRequest) request).setPassword(password);
                }
            }
        }
        return true;
    }

    public void completed(String mechanism, boolean authenticated) {

```

```

    }
}

```

When the `requested()` method is called, it makes a `PasswordRequest` and a `LogonEntryRequest`. Since only passwords are provided, the `LogonEntryRequest` has been ignored. The sample above has created a `HashMap` that contains two user names and two passwords. When the `requested()` callback method is called, it contains the user name that can be used to obtain the password from the `HashMap`. If the password matches the password provided by the client, the authentication is completed. If a password was not provided, the client is asked to once again provide a user name and password. If the service is unable to obtain the password from the user name, it requests the user name and password again.

### Code 33. SRP Logon AuthListener (client)

```

class SrpLogonListener implements AuthListener {

    static class SrpLogonListener implements AuthListener {
        public boolean requested(String mechanism,
                                String peerName,
                                int count,
                                String userName,
                                AuthRequest[] requests) {

            /*
             * Collect the requests we're interested in to simplify
             * processing below.
             */
            PasswordRequest passwordRequest = null;
            UserNameRequest userNameRequest = null;
            for (AuthRequest request : requests) {
                if (request instanceof PasswordRequest) {
                    passwordRequest = (PasswordRequest) request;
                } else if (request instanceof UserNameRequest) {
                    userNameRequest = (UserNameRequest) request;
                }
            }

            if (count <= 3) {
                System.out.print("Please enter user name [user1]: ");
                Scanner in = new Scanner(System.in);
                String user = in.nextLine();

                System.out.print("Please enter password [password1]: ");
                String password = in.nextLine();

                userNameRequest.setUserName(user);
                passwordRequest.setPassword(password.toCharArray());
                return true;
            }
        }
    }
}

```

```
        return false;
    }

    public void completed(String authMechanism,
                          String authPeer,
                          boolean authenticated) {
        if (!authenticated) {
            System.out.println("Authentication failed.");
        }
    }
}
```

The `requested()` method requests both the `PasswordRequest` and the `UserNameRequest`. Here we have asked the user to enter both user name and password. Since this is sample code, we are also revealing the user name and password which normally would not be done unless it is a guest account. If the supplied user name and password match the user name and password at the server, the authentication completes. If the user name and password do not match, the user gets up to three tries to enter the correct user name and password before the authentication fails.

## 5.10.5 Code for RSA key exchange

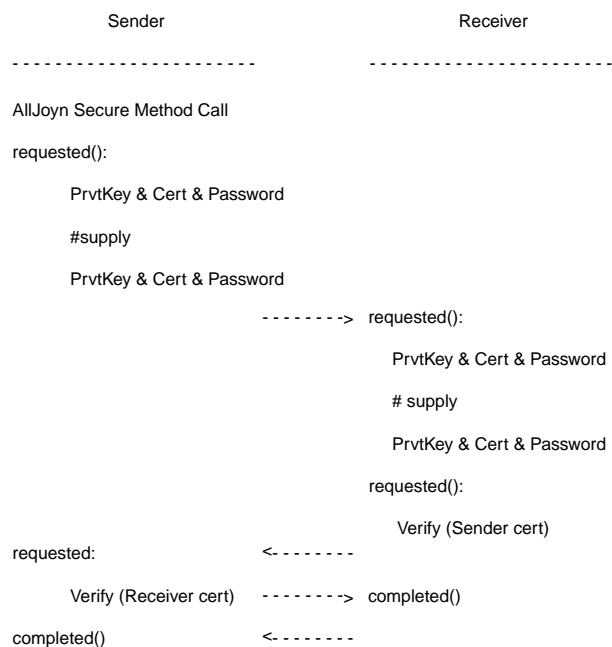
RSA is an encryption method that has been successfully used for online commerce for years. It consists of the exchange of what is called a public key. The public key can be used to encrypt data but cannot be used to decrypt the data. To be able to decrypt the data, the receiver would use their private key. If the data were encrypted using their public key, they can read the data contents but no one else can. To make private keys even more secure, they are typically encrypted using a password. To decrypt the private key, the password must be known.

The RSA key exchange certificate chain requires three things:

- A private key
- A password used to encrypt the private key
- An X.509 certificate (a standard public key structure)

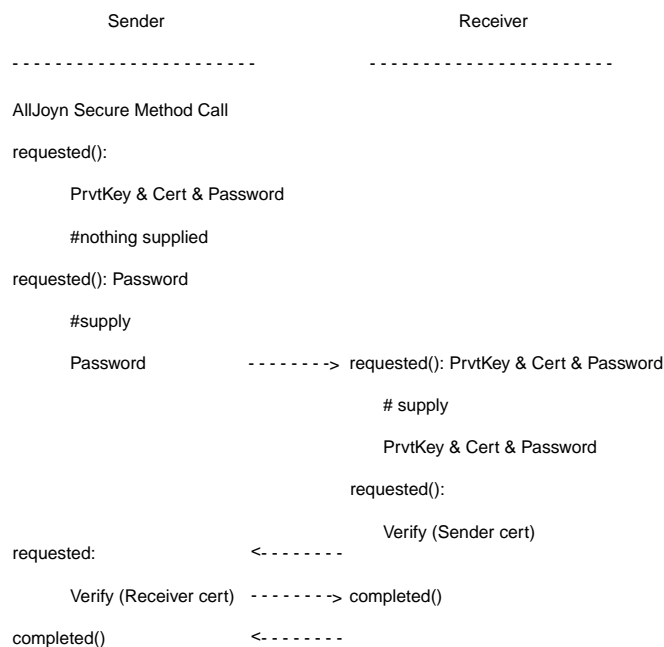
The X.509 certificate contains the public key as well as additional information such as who issued the key and how long the key is valid.

If you have the three required elements, you can provide them to the `AuthListener` when they are `requested()`. If you have not generated the required elements, the code can automatically generate them for you.



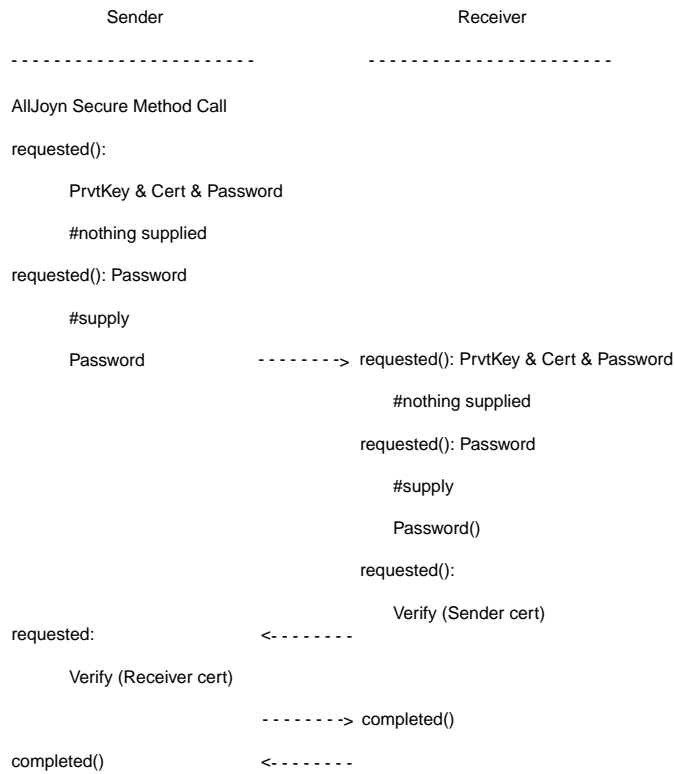
**Figure 6: MSC 6 RSA Key Exchange option 1**

For this option both sender and receiver have all the full RSA certificate chain needed: their own private keys, certificates, and the password used for private key encryption. When the sender calls the remote method call to an unverified receiver, the sender is asked to provide the RSA certificate chain. Once the RSA certificate chain has been provided, the receiver also requests the RSA certificate chain. The receiver then sends out a verify request. This gives the receiver an opportunity to inspect the certificate sent from the sender. The receiver can inspect the certificate and accept or reject the certificate based on information such as who issued the certificate.



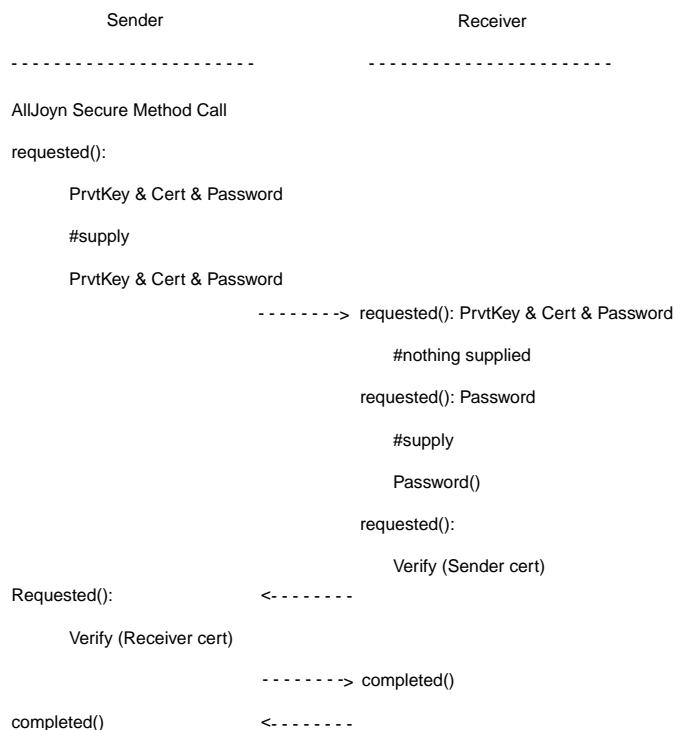
**Figure 7: MSC 7 RSA Key Exchange option 2**

For this option the sender does not have the RSA certificate chain needed for RSA authentication. The code can generate a self-signed RSA private key and X.509 certificate for the user. The user only needs to supply the password needed to encrypt the private key. If multiple authentication requests are made, the sender reuses the private key and certificate generated. It is important that each additional request for password uses the same password. If a different password is supplied, you receive an error stating that you are unable to decrypt the RSA private key. It is possible to find out if this is the first time the password is being generated by calling `AuthListener.PasswordRequest.isNewPassword()` method.



**Figure 8: MSC 8 RSA Key Exchange option 3**

In this option, neither the sender nor the receiver have the RSA elements or they were already generated. Both sides ask for their private key password. The sender and receiver can choose differing passwords since the password is only used for their own private key which is not shared with the other device.

**Figure 9: MSC 9 RSA Key Exchange option 4**

With this option the receiver side generates its own private key and certificate.

When using a secure interface, an authentication listener must be created that handles connection authentication. The authentication listener decides if the connecting client is a trusted client and if they should be allowed to use the interface.

**Code 34. Registering the authentication listener**

```

BusAttachment mBus;
mBus = new BusAttachment( "MyAppName", BusAttachment.RemoteMessage.Receive);

RsaKeyXListener mAuthListener;

Status status = mBus.registerAuthListener("ALLJOYN_RSA_KEYX",
                                          mAuthListener);

if (status != Status.OK) {
    finish();
    return;
}
  
```

The rest of the steps for connecting and advertising the service have been covered in [Discovery and sessions](#) and are unchanged for a secure interface.



This next code snippet demonstrates the auth listener for [Figure 8](#) and would work for both the service and client.

### Code 35. RSA key exchange AuthListener (service and client)

```
/* The RSA key exchange listener uses X.509 certificates to authenticate. */
class RsaKeyXListener implements AuthListener {
    public boolean requested(String mechanism,
                            String peerName,
                            int count,
                            String userName,
                            AuthRequest[] requests) {

        /*
         * Collect the requests we're interested in to simplify
         * processing below.
         */
        PasswordRequest passwordRequest = null;
        CertificateRequest certificateRequest = null;
        VerifyRequest verifyRequest = null;

        for (AuthRequest request : requests) {
            if (request instanceof PasswordRequest) {
                passwordRequest = (PasswordRequest) request;
            } else if (request instanceof CertificateRequest) {
                certificateRequest = (CertificateRequest) request;
            } else if (request instanceof VerifyRequest) {
                verifyRequest = (VerifyRequest) request;
            }
        }

        if (verifyRequest != null) {
            /* Verify a certificate chain supplied by the peer. */
            return true;
        } else if (certificateRequest != null) {
            /*
             * The engine is asking us for our certificate chain.
             *
             * If we return true and do not supply the certificate chain,
             * then the engine will create a self-signed certificate for
             * us. It will ask for the passphrase to use for the private
             * key via a PasswordRequest. If we return false authentication
             * will fail.
             */
            return true;
        } else if (passwordRequest != null) {
            /*
             * A password request under the ALLJOYN_RSA_KEYX mechanism is
             * for the passphrase of the private key.
             *
             * PasswordRequest.isNewPassword() indicates if the engine
             * needs to create a private key for us (as part of creating a

```

```

        * self-signed certificate). Otherwise it is expecting the
        * passphrase for the existing private key.
        */
        if (count <= 3) {
            if (passwordRequest.isNewPassword()) {
                System.out.print("Enter password to generate new " +
                                "private key and certificate : ");
            } else {
                System.out.print("Please enter the private key password : ");
            }
            Scanner in = new Scanner(System.in);
            String password = in.nextLine();
            passwordRequest.setPassword(password.toCharArray());
            return true;
        }
        return false;
    }

    public void completed(String mechanism,
                          String peerName,
                          boolean authenticated) {
        if (!authenticated) {
            System.out.println("Authentication failed.");
        } else {
            System.out.println("Authentication successful.");
        }
    }
}

```

When an authentication request comes in, it contains a `PrivateKeyRequest`, `CertificateRequest`, and `PasswordRequest`. This code does not provide the private key or the X.509 certificate but still returns true for the `CertificateRequest`. The code tries to see if it has already generated a private key and a self-signed certificate. If it has generated a private key and a self-signed certificate, it asks for the password that was used to generate the key and certificate. If it has not generated a private key and self-signed certificate, it asks for a password to do so. In both instances you receive a `PasswordRequest`. The only way to tell if the private key and certificate have already been generated is to check if this is a request for a new password: `passwordRequest.isNewPassword()`. It is important to remember the password, because the same password is expected on each additional authentication attempt. Both the client and the service handle RSA authentication identically, and there is no difference between the service `AuthListener` and the client `AuthListener`.

If you do not want AllJoyn generating your private key and X.509 certificate for you, then you need to provide the private key, the password used to encrypt the private key, and the X.509 certificate. For the following sample, the private key and certificate were generated

using openssl. For most systems, openssl is required to run AllJoyn. You might already have it installed on your system (openssl is not required for all installations of Windows). Many other tools are available to create the private key and certificate; if you are familiar with those tools feel free to use them.

See openssl's documentation for creating keys and certificates:

<http://www.openssl.org/docs/HOWTO/keys.txt>

<http://www.openssl.org/docs/HOWTO/certificates.txt>

To create a private key the following command was used:

```
$ openssl genrsa -des3 -out privkey.pem 2048
```

You are asked to enter a pass phrase; the pass phrase "pass" was chosen.

This generates a private key that is 2048 bits in length. 2048 or higher is recommended for RSA keys.

The private key is then used to generate the X.509 certificate

```
$ openssl req -new -x509 -key privkey.pem -out cacert.pem -days 3650
```

You are asked to provide the following for the certificate:

- The pass phrase for privkey.pem
- Country name (2 letter code)
- State or province name (full name)
- Locality name (e.g., city)
- Organization name
- Organization unit name
- Common name (e.g., YOUR name)
- Email address

This certificate is valid for just under 10 years from the date of its creation.

Inspect the contents of the certificate using openssl:

```
$ openssl x509 in cacert.pem -noout -text
```

The following code represents the sequence shown in [Figure 6](#).

#### **Code 36. RSA Key Exchange AuthListener (service and client)**

```
/* The RSA key exchange listener uses X.509 certificates to authenticate. */
private static class RsaKeyXListener implements AuthListener {
    /* password used when generating the private key */
    private static final String mPassword = "pass";
    /* 2048 bit RSA private key */
    private static final String mPrivateKey =
        "-----BEGIN RSA PRIVATE KEY-----\n" +
```

```

"Proc-Type: 4, ENCRYPTED\n" +
"DEK-Info: DES-EDE3-CBC, 61A8FD1CE34DE29B\n" +
"\n" +
"uIG/qraR9tvxhxyMPsLAP8ZimZBHMwJuMnRkAoU/fO/REnX7Zo+OLD+/ks4MnmBX\n" +

"1ziBBP3MSRcvF47C8U5qmdHPg4ujzJgbX0F6tCMOmjdeAt+iFOjov43gfuGXY3sr\n" +

"L/28OMUqrgyYVuaytVPBWOSFKy9BBSO/h/pCVHcQC+2C4vGXgz3aAoM4euqXgGXT\n" +

"xVMh8wLAMfRFJHmTmb65n8LPb5j2UOLJ3MkFXXi753qwLfqM1EO7cE5UIh+C+LwA\n" +

"KhFfehtAKMlaArnlfuJMnEuHj9vj7XfhFpdH23AXUt3CBngJvuF/xi9r0bfbwEgzi\n" +

"ONc4wKCyj2VTr3qH22ZYat5gkSh8p8qs8FsQyZE/IE7BUplhan25fevlycsvLaqZ\n" +

"zxhoCNJMrX3o+B8OqKN4sJPMte6/5BcJkvap4Wih1iNv+H0Aflc0A9Wrfww16bgx\n" +

"C3OtxfFWB8CC5d6DNxmc3Ef6M+wzbB4OW/umWThPLcqK7tjlq2hgeaOJnsW9CwXL\n" +

"o/6LoAwRss/Tp+1sw7FRKQ+cOdKXdjWwtpu8JV3EF9zibxMG00V2OLaHHU+JrOY\n" +

"j9Mkru2ItcJEOtH1hv5iNpfTMnZbgfDF1iEhAZNKDdvoSVM8MrcrQONLZ8whiUlN\n" +

"LQJU9I3oCLHJEHS23T2gMIJFyU7eb+8A+dbMEPqhvm6VK2PONEQZtQAEf+O+Hgq2\n" +

"ccx94SayajiwQoPcu8uHPEeJym2rx3/IZhttmvAqxX9GzydvsPnQedRT31PjrmQ\n" +

"bZYdNMV2Y9+GLef/x6dwFMjzHZC2o8nglCJ/PYpebISPJLRsdYt/HuGr/XFAKpQv\n" +

"jN1xbUWOYSiws/1xnidb6SYQEPHvsutaerplHeZ7XQkddKeXoz0BTF9XnY3magCd\n" +

"tWnuZtznd56pggNY4skUMF8bh/sbHtF0/MfI5SOAPmrvGsC/6+1ZG/HGM1LM9u5+\n" +

"016U1T54DjV3yDgLen1rQ4R8wZ3DLc15VVX5L4ORKqFb3073oX+RGUeS0c5x+Pqd\n" +

"/rp9CdWN3FcyC/NXsCo1GVR3qW/5/HwimQxghLqEgholZELIZskMP0zzBSejcUFJ\n" +

"5vLiusCU8C0DdcV3sfye/8kNcIalIyBeMJvia9s4AOylZsXdJyADHpHdrMsIa0R0\n" +

"FyqZBHilUMlHDbQgd0+Nkuh+zP4iXh89MJQ17Kn8Y/aqC08LYGNLaXv8lOFnqYr\n" +

"mYnR9HxLHfkKoe8qoKQqwDDrzIZGRhMYBAX2RgxSXSkfUSem9ccJ+t6vm7js5ORj\n" +

"IT8JunaRFxBYkP9popYEjV0iEKWeshyulQHpScnncca0IqkHdyNfrK+zJhoi2KhU\n" +

"eTlael/ACxPlGEZDd2pPTzwB/KdcoQQRjfi/zFovIOYmcIo1Q2eJZLgJyc3x1ne/\n" +

"21/pYZ3FiOjr3tP9MHLg3tcdi7ZShzd81bvudFCuSI569MJpGG1LYuBn/+UvohBK\n" +

"igOnHn/R60euBdsQdAxdWJqzVDm1q5J0jmN5hk2RvQ1fbX/L+cZvJsClhsO4ZX7\n"

```

HT80-BA002-1 Rev. J

```

        "UIGMr0DQryQcHFmu2m3uxoS5B7Mj1E9VwmOEKDjoxsmMQ9m/ajvj/7rT8g5/z1+m\n" +
        "N9Wk8yx0gDBJJqgmVCh+WYgniiMhfKMC+nCS9fjj9RJVumRndD/B38kpj0TyfyBt\n" +

        "e4BgMacL0w==\n" +
        "-----END CERTIFICATE-----";
    public boolean requested(String mechanism, String peerName, int count,
        String userName, AuthRequest[] requests) {
        System.out.println(String.format("AuthListener.requested(%s, %s, %d,
%s, %s);",
            mechanism,
            peerName,
            count,
            userName,
            AuthRequestsToString(requests)));

        /* Collect the requests we're interested in to simplify processing
below. */
        PasswordRequest passwordRequest = null;
        PrivateKeyRequest privateKeyRequest = null;
        CertificateRequest certificateRequest = null;
        VerifyRequest verifyRequest = null;

        for (AuthRequest request : requests) {
            if (request instanceof PasswordRequest) {
                passwordRequest = (PasswordRequest) request;
            } else if (request instanceof PrivateKeyRequest) {
                privateKeyRequest = (PrivateKeyRequest) request;
            } else if (request instanceof CertificateRequest) {
                certificateRequest = (CertificateRequest) request;
            } else if (request instanceof VerifyRequest) {
                verifyRequest = (VerifyRequest) request;
            }
        }

        if (verifyRequest != null) {
            /* Verify a certificate chain supplied by the peer. */
            return true;
        } else if (certificateRequest != null &&
            privateKeyRequest != null &&
            certificateRequest != null ) {
            /*
             * The engine is asking us for our certificate chain.
             *
             * We are providing the full certificate chain - the password,
             * private key, and certificate.
             */
            passwordRequest.setPassword(mPassword.toCharArray());
            privateKeyRequest.setPrivateKey(mPrivateKey);

```

```

        certificateRequest.setCertificateChain(mCert);

        return true;
    }
    return false;
}

public void completed(String mechanism, String peerName, boolean
authenticated) {
    if (!authenticated) {
        System.out.println("Authentication failed.");
    } else {
        System.out.println("Authentication successful.");
    }
}
}

```

When the initial request for the password, private key, and certificate come in this time, we are providing the information.

The code above simply returns true when the `verifyRequest` is received. The full process of verifying the certificate is beyond the scope of this document. The following code shows how to gain access to the certificate so that it can be verified.

### Code 37. Obtaining the X.509 certificate for verification

```

private CertificateFactory factory;
factory = CertificateFactory.getInstance("X.509");

private CertPathValidator validator;
validator = CertPathValidator.getInstance("PKIX");

if (request instanceof VerifyRequest) {
    try {
        String chain = ((VerifyRequest) request).getCertificateChain();

        BufferedInputStream in =
            new BufferedInputStream(new ByteArrayInputStream(chain.getBytes()));

        List<X509Certificate> list = new ArrayList<X509Certificate>();
        while (in.available() > 0) {
            list.add((X509Certificate) factory.generateCertificate(in));
        }

        CertPath path = factory.generateCertPath(list);

        PKIXParameters params = new PKIXParameters(trustAnchors);
        params.setRevocationEnabled(false);

        validator.validate(path, params);
    }
}

```

```
    } catch (Exception ex) {  
        return false;  
    }  
    return true;  
}
```

This code obtains the certificate as part of the `VerifyRequest`. The `chain` is converted to an `X509Certificate` that can be used to get the certificate path. The certificate path can be validated against a list of `trustedAnchors` (not shown in this code). All of our samples used self-signed certificates so there is no certificate path of authority. You would have to add yourself to the list of trusted anchors if this method of verification is used.

There are many things that could be checked in the `VerifyRequest` other than checking the certificate path of authority. For example, you could reject any certificate that was not valid for more than 10 days.

### 5.10.6 The default keystore listener

So far we have not discussed where the keys exchanged during the authentication process are stored. If we register the `AuthListener` using the two-parameter method, we are using the default keystore listener with its default options. This uses a default location for storing the keystore. The name of the keystore is the same as the application name that was used when creating the new `BusAttachment`. The default location for the keystore differs depending on the operating system that is being used. The default location chosen is a location that the user has permission to read and write from.

For Windows XP the file is in:

```
C:\Documents and Settings\<user_name>\.alljoyn_keystore\<Application name>
```

For Windows 7 the file is in:

```
C:\Users\<user_name>\.alljoyn_keystore\<Application name>
```

For Linux the file is in:

```
/home/<user_name>/.alljoyn_keystore/<Application name>
```

The default location and name of the keystore file can be changed by providing a third parameter. The third parameter can be used to specify the location where the keystore is written. Note that the location provided as the third parameter when registering the `AuthListener` is appended to the location of the home folder for the OS being used. The code must have read/write permissions to access that location.



**Code 38. Overriding the default file location when registering an AuthListener**

```
BusAttachment bus;
bus = new BusAttachment("MyApp", BusAttachment.RemoteMessage.Receive);
String strFilePath = File.separator + "foo" + File.separator + "fileName.ks";
Status status = mBus.registerAuthListener("ALLJOYN_SRP_KEYX",
                                          mAuthListener,
                                          strFilePath);
```

This places the file in the users home directory in a folder named foo. The file is called fileName.ks.

C:\Documents and Settings\\foo\fileName.ks (Windows XP)

C:\Users\\foo\fileName.ks (Windows 7)

/home/<user\_name>/foo/fileName.ks (Linux)

**Note** The three-parameter option should be used when registering an authListener in Android (see [Security](#)).

Multiple applications can share a single keystore; however, the code needs to know that the keystore is being shared between multiple applications or it overrides changes made by other applications. To use a shared keystore you must specify that the keystore is shared when registering the authListener.

**Code 39. Registering AuthListeners with a shared keystore**

```
BusAttachment bus;
bus = new BusAttachment("MyApp", BusAttachment.RemoteMessage.Receive);
String strFilePath = File.separator + ".alljoyn_keystore" + File.separator +
                      "shared.ks";
Status status = mBus.registerAuthListener("ALLJOYN_SRP_KEYX",
                                          mAuthListener,
                                          strFilePath,
                                          true);
```

Every application that wants to share the same keystore should point to the same keystore file and set the `isShared` option to true when registering an `AuthListener`.

Why would you want multiple applications to share a keystore file? This allows you to set up a family of applications that authenticate as one. Once one of the applications has completed authentication, then all of the applications can use the same set of keys. This means that only one authentication request is needed, even though multiple applications use a secure interface. Under AllJoyn's default mode, authentication is done on an app-per-app basis, not as a family of applications.

## 5.11 AllJoyn's debug log output

AllJoyn has a built-in logging system for investigating AllJoyn-related issues. Only critical log output is displayed when using `Release` mode of AllJoyn.

To enable AllJoyn debugging three things must be done:

1. Set the OS Logging option (OS specific)
2. Set the Debug Level for the module(s) of interest (on the daemon or the application)
3. Set `ER_DEBUG_*` environment variable (only needed when logging to `stderr`)

These can be accessed using the following method calls:

```
BusAttachment.useOSLogging (Boolean useOSLog );
BusAttachment.setDebugLevel(String module, int level);
BusAttachment.setLogLevels(String logEnv);
BusAttachment.setDaemonDebug(String module, int level);
```

### 5.11.1 Set OS-specific logging option

`BusAttachment.useOSLogging` is a command that results in different behavior depending on what OS the AllJoyn code is running on, as shown in [Table 6](#).

**Table 6: OS-specific logging behavior**

OS	True	False
Windows	Debugger Trace Log	stderr
Windows RT	Win RT Trace Log	Win RT File Log
Linux	stderr	stderr
Android	LogCat	stderr
Mac	stderr	stderr

For `BusAttachment.useOSLogging(true)` on Windows and Windows RT see:

`OutputDebugString` function (Windows)

[http://msdn.microsoft.com/en-us/library/windows/desktop/aa363362\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa363362(v=vs.85).aspx)

For `BusAttachment.useOSLogging(false)` on Windows RT, a file named `alljoyn.log` will be created in the user's `Documents` directory.

### 5.11.2 Set Debug Level for modules

AllJoyn has a large number of modules, with new modules being added as AllJoyn continues to be developed. Several Module names of interest are:

- ALL – print all modules
- ALLJOYN\_JAVA – trace log for the JNI layer of the Java Bindings
- ALLJOYN – trace of AllJoyn method calls

- ALLJOYN\_OBJ – trace of AllJoyn to name service calls
- NS – the TCP name service

Each module can print out four debug levels:

- High Level = 0b0001
- Debug Messages = 0b0010
- API Trace = 0b0100
- Data Dump = 0b1000

As a general rule you want a debug level and all the debug options below that option. For example:

1 = “High Level”

3 = “High Level” and “Debug Messages”

7 = “High Level”, “Debug Messages” and “API trace”

15 = “High Level”, “Debug Messages”, “API trace” and “Data Dump”

The following code would print the High Level log for ALL of the AllJoyn modules and the API trace and below log for the ALLJOYN module:

```
BusAttachment.setDebugLevel("ALL", 1);
BusAttachment.setDebugLevel("ALLJOYN", 7);
```

The same could be achieved using the following code:

```
BusAttachment.setLogLevels("ALL=1;ALLJOYN=7");
```

If your application is using a separate AllJoyn daemon or it is using the built-in name discovery/advertisement services, the logs from this part of AllJoyn will not output logs unless you use the following call:

```
BusAttachment.setDaemonDebug("ALL", 7);
```

**Note** There is an error debug level that is always printed and cannot be turned off.

### 5.11.3 Set ER\_DEBUG\_\* environment variable

When logging to `stderr`, an environment variable must also be specified to obtain the debug log. This is a consequence of the AllJoyn logging system being written in C++.

Normally when using a C++ program the user would only need to specify the module they are interested in using environment variables. For example:

```
export ER_DEBUG_ALLJOYN_OBJ=15
```

With Java you will generally use:

- Linux

```
export ER_DEBUG_ALL=15
```

## ■ Windows

```
set ER_DEBUG_ALL=15
```

### 5.11.4 Log output

If everything is set up as expected you should be log output as follows:

```
6710.723 DEBUG    THREAD    external    ...n\os\windows\Thread.cc:352 |
    Thread::Join() thread is dead [replyTimer]
6710.724 HL_DBG   THREAD    external    ...n\os\windows\Thread.cc:185 |
    Thread::~Thread() [replyTimer,738e6b0] started:3 running:3 stopped:0
6710.725 DEBUG    ALLJOYN_JAVA external    ...a\jni\alljoyn_java.cc:3942 |
    JBusAttachment::Connect(): Taking Bus Attachment authentication listener
    change lock
6710.742 DEBUG    ALLJOYN_JAVA external    ...a\jni\alljoyn_java.cc:4145 |
    JBusAttachment::EnablePeerSecurity()
6710.743 TRACE    ALLJOYN    external    ...e\src\BusAttachment.cc:359 |
    BusAttachment::TryConnect to tcp:addr=127.0.0.1,port=9956
6710.744 HL_DBG   ALLJOYN    external    ...ows\ClientTransport.cc:136 |
    ClientTransport::Connect(): tcp:addr=127.0.0.1,port=9956
Initialized winsock
6710.764 TRACE    NETWORK    external    ...n\os\windows\Socket.cc:164 |
    Socket(addrFamily = 2, type = 1, sockfd = <>)
6710.766 TRACE    NETWORK    external    ...n\os\windows\Socket.cc:188 |
    Connect(sockfd = 612, remoteAddr = 127.0.0.1, remotePort = 9956)
6711.827 HL_DBG   ALLJOYN    external    ...ows\ClientTransport.cc:175 |
    ClientTransport(): socket Connect() failed ER_CONN_REFUSED
6711.828 TRACE    NETWORK    external    ...n\os\windows\Socket.cc:361 |
    Close (sockfd = 612)
6711.831 TRACE    ALLJOYN    external    ...e\src\BusAttachment.cc:359 |
    BusAttachment::TryConnect to null:
Using BundledDaemon
```

There are five columns for each log message printed out:

1. Time stamp representing when the log was printed.
2. The debug level.
3. The AllJoyn module the message came from.
4. Which thread the module was running on.
5. Which line of source code printed the message “|” the debug message.

## 6 Android-Specific Considerations

---

### 6.1 UI thread vs. AllJoyn bus thread

When Android starts a new activity, a thread is created that draws all of the user interface elements on the screen. By default, all code that is added to the activity also runs in the same thread. Android always expects the user interface (UI) to be responsive. If a function is called that takes too long to process information, the application is terminated by Android simply because the UI was unresponsive.

AllJoyn applications typically conform to the client/server model; the application is either providing a service or requesting something from a service. The service could be on a remote device or on the same device. There are no time guarantees when calling an AllJoyn method. For this reason, all calls to an AllJoyn method or signal must be done in a thread that is independent of the main UI thread.

This could be done by creating a new thread each time you call a signal or method associated with AllJoyn. This can result in many threads being created including all the memory usage associated with each thread. If too many threads are created, the application could quickly run out of memory and crash. This could easily happen if signals were being used to constantly update something like a game.

To avoid using a separate thread for each AllJoyn call, we suggest you create a Handler on a separate thread that queues AllJoyn method and signal calls.

### 6.2 Using Handler Looper

Create an implementation of a Handler that takes a Looper as its initialization parameter.

#### **Code 40. BusHandler class that extends the Handler class**

```
class BusHandler extends Handler {
    public BusHandler(Looper looper) {
        super(looper);
    }

    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            default:
                break;
        }
    }
}
```

```
}  
}
```

Now all of the AllJoyn method calls and signal calls can be placed inside the switch statement found in the `handleMessage` method.

Almost all programs that use AllJoyn need to implement a connect case and a disconnect case, as well as a case for each AllJoyn method, property, and signal.

To start the `BusHandler`, use the following code:

#### Code 41. Starting the `BusHandler` class

```
BusHandler mBusHandler;  
HandlerThread busThread = new HandlerThread("BusHandler");  
busThread.start();  
mBusHandler = new BusHandler(busThread.getLooper());
```

The `<AllJoyn_dist>/java/samples` folder contains a large number of samples that all show how to use the `BusHandler`.

## 6.3 Security

In the AllJoyn Java API, there are two `BusAttachment.registerAuthListener( ... )` methods. One takes two arguments, and the other takes three arguments. When registering the authentication listener, the one with three arguments should be used. The third argument sets the name of the keystore. It is recommended to always set this to `getFilePathStreamPath("alljoyn_keystore").getAbsolutePath()` for all Android programs.

## 6.4 Peer Permission Check

In Android's security model, each application must request permission to access certain system resources. For example, `android.permission.READ_CONTACTS` is required to access the contact information on an Android device.

In catering to the security model of Android, AllJoyn allows the service to impose permission requirements on an AllJoyn Method/Signal call. If the service specifies the permission requirements on a call, the client must acquire those permissions by declaring them in the `AndroidManifest.xml` in order to invoke the call. If a client has not added the proper permissions to the `AndroidManifest.xml`, the AllJoyn Method/Signal call made by the client is rejected. If it is an AllJoyn Method call, the daemon sends back an error message (`status = ER_ALLJOYN_ACCESS_PERMISSION_ERROR`) to the caller. Multiple permissions should be concatenated into a single string using a semicolon (;) as the delimiter, for example:

```
android.permission.X;android.permission.Y
```

By using the access permission in the interface, we are preventing a client application from accessing data without also requesting the same device permission level as the service.

**Important** The peer permission check model only applies when the service/client are on the same device. If the service and client are on different devices, the peer permission check will have no effect. The device-to-device trust model should be based on Authentication.

Android's security model does not apply to other OSs. If AllJoyn is used to synch contact and calendar data with a desktop computer, the computer is not expected to obtain permission to read and write the calendar and contacts. The application installed on the Android device that is communicating with the computer would still be required to have Android permissions to access the information of interest.

#### Code 42. Adding the `@AccessPermission` annotation for an AllJoyn Java interface

```
@BusInterface (name = "org.alljoyn.bus.addressbook")
public interface AddressBookInterface {
    /*
     * The AccessPermission annotation signifies that this function uses
     * system resources that require system permission
     * "android.permission.READ_CONTACTS". Thus
     * the application should declare the required permissions in its
     * manifest file. If there are more than one permission item,
     * they should be separated by ';'.
     */
    @BusMethod(signature = "si", replySignature = "r")
    @AccessPermission("android.permission.READ_CONTACTS")
    public Contact getContact(String name, int userId) throws BusException;

    /*
     * the replySignature indicates that this BusMethod will return an
     * array of structs.
     */
    @BusMethod(replySignature = "ar")
    @AccessPermission("android.permission.READ_CONTACTS")
    public NameId[] getAllContactNames() throws BusException;
}
```

## 6.5 Bundled Daemon (Deprecated)

If you are using AllJoyn 2.6, the concept of bundled daemon is no longer present. A developer can still get all the functionality that was available with bundled daemon in AllJoyn 2.3.6 but there is no requirement to add the additional library (bundle.jar) to the Android app or to use the extra calls that were explicitly needed to use bundle daemon. However, the function call

`org.alljoyn.bus.alljoyn.DaemonInit.PrepareDaemon(getApplicationContext());` is now used for an entirely different purpose as explained in [Using ICE with AllJoyn](#).

## 6.6 Using ICE with AllJoyn

AllJoyn does what it is supposed to do; wonderfully. However, it cannot control certain things like the network setup over which it talks. A common example is wireless isolation. What is wireless isolation?

Imagine yourself sitting in a coffee shop and wanting to play a game with your friend that uses AllJoyn. In a normal home setup you would be connected to your home access point and this would work seamlessly. In a coffee shop or public place the network administrators set up the access points such that two peers cannot talk to each other directly. Things like multicast and broadcast are disabled in the coffee shop to avoid bandwidth problems for other users – something that could happen if a young college student were to set up his own peer-to-peer network in the coffee shop with his friends or use an application (not AllJoyn) which floods the network with multicast/broadcast packets.

AllJoyn's philosophy is to make it unnecessary for the developer to be aware of such network complexities and still achieve what AllJoyn originally delivers – easy service discovery and connectivity without having to change anything depending on the network or platform. To overcome the problem of wireless isolation, we added in a parallel transport in AllJoyn that we call ICE. ICE makes sure that you are able to discover services/devices and connect to them even if the access point you are connected to does not allow this to happen directly through it.

ICE is not only a way to solve the wireless isolation problem, it also provides another means to detect proximity with other devices to aid in service discovery. To do this ICE needs to know the list of access points around you. It does this by talking to the Android framework from your AllJoyn app. For this reason you need to specify the call:

```
org.alljoyn.bus.alljoyn.DaemonInit.PrepareDaemon(getApplicationContext());
```

An ideal place to call this is just before you create a `BusAttachment` in your Android application. The application context is required so that the internal AllJoyn Java framework can get a list of access points around you by talking to the Android framework.

## 6.7 Permissions to Add to your AllJoyn Android App

Under the hood, AllJoyn does a lot of different things on different platforms for device discovery without making the developer aware of the complexities involved. On Android, it requires that the developer of the app grant the app a set of four permissions so that the AllJoyn discovery works seamlessly.

The two permissions below are required because traditionally AllJoyn uses multicast for device discovery and the Android framework requires any app that intends to use multicast to have these permissions.

```
<uses-permission  
android:name="android.permission.INTERNET"></uses-permission>
```



```
<uses-permission
android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE"></uses-permission>
```

The two permissions below are required when you do not want to worry about wireless isolation that may be present on the network to which you are connected. These two permissions let AllJoyn deal with it using ICE. These permissions enable AllJoyn to look for wireless access points around you and use this data to detect if you are in proximity to any other device.

```
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
```

```
<uses-permission
android:name="android.permission.CHANGE_WIFI_STATE"></uses-permission>
```

## 6.8 ProGuard

ProGuard is one of the tools packaged with the Android SDK. ProGuard is used to shrink, optimize, and obfuscate code when producing a release version of an .apk file. After using ProGuard, the resulting .apk is smaller and more difficult to reverse engineer. For more information, see <http://developer.android.com/guide/developing/tools/proguard.html>

Knowing the names of methods, interfaces, annotations, and classes is important for AllJoyn to operate properly. If the code is passed through ProGuard the names will be changed, annotations will be removed and AllJoyn code will no longer function as expected. To use ProGuard, the proguard.cfg file must have additional options added.

Below is a sample proguard.cfg file that would be needed if we wanted to use ProGuard with the Android Chat sample. The proguard.cfg is based on the default file created when using an Android project. In the sample proguard.cfg file below, the text in bold is an example of what must be added.

### Code 43. Sample proguard.cfg file

```
-optimizationpasses 5
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-dontpreverify
-verbose
-optimizations !code/simplification/arithmetic,!field/*,!class/merging/*

#REQUIRED FOR ALLJOYN TO FUNCTION WHEN USED WITH PROGUARD
# (NOTE: could use -keep class org.alljoyn.bus** {*; } but the
# alljoyn samples use org.alljoyn.bus.samples so if you checked the .apk
# obfuscated code would not be seen)
-dontwarn org.alljoyn.bus.**
-keepattributes *Annotation*
-keepattributes Signature
-keep class org.alljoyn.bus.annotation.** { *; }
```

```

-keep class org.alljoyn.bus.ifaces.** { *; }
-keep class org.alljoyn.bus.AnnotationBusException** { *; }
-keep class org.alljoyn.bus.AuthListener** { *; }
-keep class org.alljoyn.bus.BusAttachment** { *; }
-keep class org.alljoyn.bus.BusException** { *; }
-keep class org.alljoyn.bus.BusListener** { *; }
-keep class org.alljoyn.bus.BusObject** { *; }
-keep class org.alljoyn.bus.BusObjectListener** { *; }
-keep class org.alljoyn.bus.ErrorReplyBusException** { *; }
-keep class org.alljoyn.bus.InterfaceDescription** { *; }
-keep class org.alljoyn.bus.IntrospectionListener** { *; }
-keep class org.alljoyn.bus.KeyStoreListener** { *; }
-keep class org.alljoyn.bus.MarshalBusException** { *; }
-keep class org.alljoyn.bus.MessageContext** { *; }
-keep class org.alljoyn.bus.MsgArg** { *; }
-keep class org.alljoyn.bus.Mutable** { *; }
-keep class org.alljoyn.bus.OnJoinSessionListener** { *; }
-keep class org.alljoyn.bus.ProxyBusObject** { *; }
-keep class org.alljoyn.bus.SecurityViolationListener** { *; }
-keep class org.alljoyn.bus.SessionListener** { *; }
-keep class org.alljoyn.bus.SessionOpts** { *; }
-keep class org.alljoyn.bus.SessionPortListener** { *; }
-keep class org.alljoyn.bus.SignalEmitter** { *; }
-keep class org.alljoyn.bus.Signature** { *; }
-keep class org.alljoyn.bus.Status** { *; }
-keep class org.alljoyn.bus.Variant** { *; }
-keep class org.alljoyn.bus.VariantTypeReference** { *; }
#Keep all BusObjects
-keep class * implements org.alljoyn.bus.BusObject { *; }
-keepclassmembers class * implements org.alljoyn.bus.BusObject {*; }
#-----
#USER NEEDS TO MODIFY WITH VALUES FROM THEIR APPLICATION - CHAT USED AS AN
EXAMPLE
#Need to keep all AllJoyn interfaces and classmembers of the AllJoyn interfaces
-keep interface org.alljoyn.bus.sample.chat.ChatInterface
-keepclassmembers interface org.alljoyn.bus.sample.chat.ChatInterface { *; }
#Need to keep the signal methods
-keepclassmembers class org.alljoyn.bus.sample.chat.AllJoynService {
    public void Chat(java.lang.String);
}
#-----
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider
-keep public class * extends android.app.backup.BackupAgentHelper
-keep public class * extends android.preference.Preference
-keep public class com.android.vending.licensing.ILicensingService

```

```
-keepclasseswithmembernames class * {
    native <methods>;
}

-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet);
}

-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet, int);
}

-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

-keep class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator *;
}
```

# 7 Performance

---

## 7.1 Header compression

When a signal is repetitively emitted, the signal header could end up being a significant portion of the data that is sent. This is especially true when the contents of the signal are really small.

Header compression can be used to improve performance. Since setting up header compression is an expensive operation, it should not be used except for repetitive signals.

An example would be a game that wants to constantly update its player's positions. The same signal would be repetitively sent with the new player position. Header compression could be used to reduce the size of the signal.

### Code 44. Header compression

```
// Create the signal emitter and connect it with the SampleInterface class
SignalEmitter emitter = new SignalEmitter(mySignalInterface,
                                           SignalEmitter.GlobalBroadcast.On);
emitter.setCompressHeader(true);
```

## 7.2 Shutting off discovery

When trying to transmit large amounts of data via Bluetooth, discovery can significantly slow down the transfer speed. If the transmit time is a factor, it may be beneficial to stop discovery when sending the data.

It is good practice to shut off discovery when an interface has been found even when using TCP/IP. Leaving discovery running could have an impact on battery life and network performance since the daemon continues to access the radio to look for remote services.

### Code 45. Shutting off Discovery

```
status = mBus.cancelFindName(SERVICE_NAME);
if (status != Status.OK) {
    finish();
    return;
}
```

## 7.3 Signal emitters

When setting up a `SignalEmitter`, you can set it up to broadcast:

- Globally to clients only on the local device

- Globally to all clients even off device
- To a session
- To a single destination

As a general rule it is best to avoid signal emitters that do not exist in the same session. This creates a large amount of network traffic. If you want the signal to be received by multiple devices, it is best to set up a multipoint session and send the signal to all the devices in the session. This reduces the network traffic by sending the signal to destinations that have registered a signal handler for the signal.