D-Bus programming in Java 1.5

Matthew Johnson
dbus@matthew.ath.cx

November 15, 2016

Contents

List of Figures

List of Tables

1 Introduction

This document describes how to use the Java implementation of D-Bus. D-Bus is an IPC mechanism which at a low level uses message passing over Unix Sockets or IP. D-Bus models its messages as either function calls on remote objects, or signals emitted from them.

Java is an object-oriented language and this implementation attempts to match the D-Bus IPC model to the Java object model. The implementation also make heavy use of threads and exceptions. The programmer should be careful to take care of synchronisation issues in their code. All method calls by remote programs on exported objects and all signal handlers are run in new threads. Any calls on remote objects may throw `DBusExecutionException`, which is a runtime exception and so the compiler will not remind you to handle it.

The Java D-Bus API is also documented in the JavaDoc[1], D-Bus is described in the specification[2] and the API documentation[3].

1.1 Protocol Implementation

This library is a native Java implementation of the D-Bus protocol and not a wrapper around the C reference implementation.

1.2 Dependencies

This library requires Java 1.5-compatible VM and compiler (either Sun, or ecj+jamvm with classpath-generics newer than 0.19) and the unix socket, debug and hexdump libraries from `http://www.matthew.ath.cx/projects/java/`.

1.3 D-Bus Terminology

D-Bus has several notions which are exposed to the users of the Java implementation.

[1] `http://dbus.freedesktop.org/doc/dbus-java/api/`
[2] http://dbus.freedesktop.org/doc/dbus-specification.html
[3] http://dbus.freedesktop.org/doc/api/html/

1.3.1 Bus Names

Programs on the bus are issued a unique identifier by the bus. This is guaranteed to be unique within one run of the bus, but is assigned sequentially to each new connection.

There are also so called well-known bus names which a device can request on the bus. These are of the form *"org.freedesktop.DBus"*, and any program can request them if they are not already owned.

1.3.2 Interfaces

All method calls and signals are specified using an interface, similar to those in Java. When executing a method or sending a signal you specify the interface the method belongs to. These are of the form *"org.freedesktop.DBus"*.

1.3.3 Object Paths

A program may expose more than one object which implements an interface. Object paths of the form *"/org/freedesktop/DBus"* are used to distinguish objects.

1.3.4 Member Names

Methods and Signals have names which identify them within an interface. D-Bus does not support method overloading, only one method or signal should exist with each name.

1.3.5 Errors

A reply to any message may be an error condition. In which case you reply with an error message which has a name of the form *"org.freedesktop.DBus.Error.ServiceUnknown"*.

2 DBusConnection

The `DBusConnection`[4] class provides methods for connecting to the bus, exporting objects, sending signals and getting references to remote objects.

`DBusConnection` is a singleton class, multiple calls to `getConnection` will return the same bus connection.

```
conn = DBusConnection.getConnection(DBusConnection.SESSION);
```

This creates a connection to the session bus, or returns the existing connection.

```
conn.addSigHandler(TestSignalInterface.TestSignal.class,
                   new SignalHandler());
```

This sets up a signal handler for the given signal type. SignalHandler.handle will be called in a new thread with an instance of TestSignalInterface.TestSignal when that signal is recieved.

[4] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusConnection.html`

```
conn.sendSignal(new TestSignalInterface.TestSignal(
                    "/foo/bar/com/Wibble",
                    "Bar",
                    new UInt32(42)));
```

This sends a signal of type `TestSignalInterface.TestSignal`, from the object *"/foo/bar/com/Wibble"* with the arguments *"Bar"* and `UInt32(42)`.

```
conn.exportObject("/Test", new testclass());
```

This exports the `testclass` object on the path *"/Test"*

```
Introspectable intro = (Introspectable) conn.getRemoteObject(
                                "foo.bar.Test", "/Test",
                                Introspectable.class);
```

This gets a reference to the *"/Test"* object on the process with the name *"foo.bar.Test"* . The object implements the `Introspectable` interface, and calls may be made to methods in that interface as if it was a local object.

```
String data = intro.Introspect();
```

The Runtime Exception `DBusExecutionException` may be thrown by any remote method if any part of the execution fails.

2.1 Asynchronous Calls

Calling a method on a remote object is synchronous, that is the thread will block until it has a reply. If you do not want to block you can use an asynchronous call.

There are two ways of making asynchronous calls. You can either call the `callMethodAsync` function on the connection object, in which case you are returned a `DBusAsyncReply`[5] object which can be used to check for a reply and get the return value. This is demonstrated in figure 1.

```
DBusAsyncReply<Boolean> stuffreply =
   conn.callMethodAsync(remoteObject, "methodname", arg1, arg2);
...
if (stuffreply.hasReply()) {
   Boolean b = stuffreply.getReply();
   ...
}
```

Figure 1: Calling an asynchronous method

Alternatively, you can register a callback with the connection using the `callWithCallback` function on the connection object. In this case, your callback class (implementing the `CallbackHandler`[6] interface will be called when the reply is returned from the bus.

[5] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusAsyncReply.html`

[6] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/CallbackHandler.html`

3 DBusInterface

To call methods or expose methods on D-Bus you need to define them with their exact signature in a Java interface. The full name of this interface must be the same as the D-Bus interface they represent. In addition, D-Bus interface names must contain at least one period. This means that DBusInterfaces cannot be declared without being in a package.

For example, if I want to expose methods on the interface *"org.freedesktop.DBus"* I would define a Java interface in the package `org.freedesktop` called `DBus`. This would be in the file `org/freedesktop/DBus.java` as normal. Any object wanting to export these methods would implement `org.freedesktop.DBus`.

Any interfaces which can be exported over D-Bus must extend `DBusInterface`[7]. A class may implement more than one exportable interface, all public methods declared in an interface which extend `DBusInterface` will be exported.

A sample interface definition is given in figure 2, and a class which implements it in figure 3. More complicated definitions can be seen in the test classes[8].

All method calls by other programs on objects you export over D-Bus are executed in their own thread.

`DBusInterface` itself specifies one method `boolean isRemote()`. If this is executed on a remote object it will always return true. Local objects implementing a remote interface must implement this method to return false.

```
package org.freedesktop;
import org.freedesktop.dbus.UInt32;
import org.freedesktop.dbus.DBusInterface;

public interface DBus extends DBusInterface
{
   public boolean NameHasOwner(String name);
   public UInt32 RequestName(String name, UInt32 flags);
}
```

Figure 2: An interface which exposes two methods

3.1 Interface name overriding

It is highly recommended that the Java interface and package name match the D-Bus interface. However, if, for some reason, this is not possible then the name can be overridden by use of an Annotation.

[7] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusInterface.html`

[8] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/test/TestRemoteInterface2.java` `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/test/TestRemoteInterface.java`

```
package my.real.implementation;
import org.freedesktop.dbus.DBus;
import org.freedesktop.dbus.UInt32;

public class DBusImpl implements DBus
{
   Vector<String> names;
   public boolean NameHasOwner(String name)
   {
      if (names.contains(name)) return true;
      else return false;
   }
   public UInt32 RequestName(String name, UInt32 flags)
   {
      names.add(name);
      return new UInt32(0);
   }
   public boolean isRemote() { return false; }
}
```

Figure 3: A class providing a real implementation which can be exported

To override the Java interface name you should add an annotation to the interface of `DBusInterfaceName`[9] with a value of the desired D-Bus interface name. An example of this can be seen in figure 4.

```
package my.package;
import org.freedesktop.dbus.DBusInterface;
import org.freedesktop.dbus.DBusInterfaceName;

@DBusInterfaceName("my.otherpackage.Remote")
public interface Remote extends DBusInterface
{
   ...
}
```

Figure 4: Overloading the name of an interface.

If you have signals which are declared in a renamed interface (see below for signals) then when adding a signal handler you *must* use an `addSigHandler` method which takes a class object corresponding to that signal. If you do not then receiving the signal will fail.

[9] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusInterfaceName.html`

4 DBusSignal

Signals are also declared as part of an interface. The Java API models these as inner classes within an interface. The containing interface must extend `DBusInterface`, and the inner classes representing the signals must extend `DBusSignal`[10]. The Signal name is derived from the name of this inner class, and the interface from its containing interface.

Signals can take parameters as methods can (although they cannot return anything). For the reflection to work, a Signal declare a single constructor of the correct type. The constructor must take the object path they are being emitted from as their first (String) argument, followed by the other parameters in order. They must also call the superclass constructor with the same parameters. A full definition of a signal can be seen in figure 5. Again, more complicated definitions are available in the test classes[11].

```
package org.freedesktop;
import org.freedesktop.dbus.DBusInterface;
import org.freedesktop.dbus.DBusSignal;
import org.freedesktop.dbus.exceptions.DBusException;

public interface DBus extends DBusInterface
{
   public class NameAcquired extends DBusSignal
   {
      public final String name;
      public NameAcquired(String path, String name)
                              throws DBusException
      {
         super(path, name);
         this.name = name;
      }
   }
}
```

Figure 5: A Signal with one parameter

5 DBusExecutionException

If you wish to report an error condition in a method call you can throw an instance of `DBusExecutionException`[12]. This will be sent back to the caller as an error message, and the error name is taken from the class name of the exception. For example, if you

[10] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusSignal.html`

[11] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/test/TestSignalInterface.html`

[12] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusExecutionException.html`

wanted to report an unknown method you would define an exception as in figure 6 and then throw it in your method as in figure 7.

```
package org.freedesktop.DBus.Error;
import org.freedesktop.dbus.exceptions.DBusExecutionException;

public class UnknownMethod extends DBusExecutionException
{
   public UnknownMethod(String message)
   {
      super(message);
   }
}
```

Figure 6: An Exception

```
...
public void throwme() throws org.freedesktop.DBus.Error.UnknownMethod
{
   throw new org.freedesktop.DBus.Error.UnknownMethod("hi");
}
...
```

Figure 7: Throwing An Exception

If you are calling a remote method and you want to handle such an error you can simply catch the exception as in figure 8.

```
...
try {
   remote.throwme();
} catch (org.freedesktop.DBus.Error.UnknownMethod UM) {
   ...
}
...
```

Figure 8: Catching An Exception

6 DBusSigHandler

To handle incoming signals from other programs on the Bus you must register a signal handler. This must implement `DBusSigHandler`[13] and provide an implementation for

[13] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusSigHandler.html`

the handle method. An example Signal Handler is in figure 9. Signal handlers should be parameterised with the signal they are handling. If you want a signal handler to handle multiple signals you can leave out the parameterisation and use `instanceof` to check the type of signal you are handling. Signal handlers will be run in their own thread.

```
import org.freedesktop.dbus.DBusSignal;
import org.freedesktop.dbus.DBusSigHandler;

public class Handler extends DBusSigHandler<DBus.NameAcquired>
{
   public void handle(DBus.NameAcquired sig)
   {
         ...
   }
}
```

Figure 9: A Signal Handler

7 D-Bus Types

D-Bus supports a number of types in its messages, some which Java supports natively, and some which it doesn't. This library provides a way of modelling the extra D-Bus Types in Java. The full list of types and what D-Bus type they map to is in table 1.

7.1 Basic Types

All of Java's basic types are supported as parameters and return types to methods, and as parameters to signals. These can be used in either their primitive or wrapper types.

7.1.1 Unsigned Types

D-Bus, like C and similar languages, has a notion of unsigned numeric types. The library supplies `UInt16`[14], `UInt32` and `UInt64` classes to represent these new basic types.

7.2 Strings

D-Bus also supports sending Strings. When mentioned below, Strings count as a basic type.

7.2.1 String Comparisons

There may be some problems with comparing strings received over D-Bus with strings generated locally when using the String.equals method. This is due to how the Strings are generated from a UTF8 encoding. The recommended way to compare strings which have

[14] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/UInt16.html`

been sent over D-Bus is with the `java.text.Collator` class. Figure 10 demonstrates its use.

```
String rname = remote.getName();
Collator col = Collator.getInstance();
col.setDecomposition(Collator.FULL_DECOMPOSITION);
col.setStrength(Collator.PRIMARY);
if (0 != col.compare("Name", rname))
   fail("getName return value incorrect");
```

Figure 10: Comparing strings with `java.text.Collator`.

7.3 Arrays

You can send arrays of any valid D-Bus Type over D-Bus. These can either be declared in Java as arrays (e.g. `Integer[]` or `int[]`) or as Lists (e.g. `List<String>`). All lists **must** be parameterised with their type in the source (reflection on this is used by the library to determine their type). Also note that arrays cannot be used as part of more complex type, only Lists (for example `List<List<String>>`).

7.4 Maps

D-Bus supports a dictionary type analogous to the Java Map type. This has the additional restriction that only basic types can be used as the key (including String). Any valid D-Bus type can be the value. As with lists, maps must be fully parameterised. (e.g. `Map<Integer, String>`).

7.5 Variants

D-Bus has support for a Variant type. This is similar to declaring that a method takes a parameter of type `Object`, in that a Variant may contain any other type. Variants can either be declared using the `Variant`[15] class, or as a Type Variable. In the latter case the value is automatically unwrapped and passed to the function. Variants in compound types (Arrays, Maps, etc) must be declared using the Variant class with the full type passed to the Variant constructor and manually unwrapped.

Both these methods use variants:

```
public void display(Variant v);
public <T> int hash(T v);
```

7.6 Structs

D-Bus has a struct type, which is a collection of other types. Java does not have an analogue of this other than fields in classes, and due to the limitation of Java reflection

[15] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/Variant.html`

this is not sufficient. The library declares a `Struct`[16] class which can be used to create structs. To define a struct you extend the `Struct` class and define fields for each member of the struct. These fields then need to be annotated in the order which they appear in the struct (class fields do not have a defined order). You must also define a single constructor which takes the contents of he struct in order. This is best demonstrated by an example. Figure 11 shows a Struct definition, and figure 12 shows this being used as a parameter to a method.

```
package org.freedesktop.dbus.test;

import org.freedesktop.dbus.DBusException;
import org.freedesktop.dbus.Position;
import org.freedesktop.dbus.Struct;

public final class TestStruct extends Struct
{
   @Position(0)
   public final String a;
   @Position(1)
   public final int b;
   @Position(2)
   public final String c;
   public Struct3(String a, int b, String c)
   {
      this.a = a;
      this.b = b;
      this.c = c;
   }
}
```

Figure 11: A Struct with three elements

```
public void do(TestStruct data);
```

Figure 12: A struct as a parameter to a method

Section 10 describes how these can be automatically generated from D-Bus introspection data.

7.7 Objects

You can pass references to exportable objects round using their object paths. To do this in Java you declare a type of `DBusInterface`. When the library receive- an object path

[16] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/Struct.html`

it will automatically convert it into the object you are exporting with that object path. You can pass remote objects back to their processes in a similar fashion.

Using a parameter of type `DBusInterface` can cause the automatic creation of a proxy object using introspection. If the remote app does not support introspection, or the object does not exist at the time you receive the message then this will fail. In that case the parameter can be declared to be of type `Path`. In this case no automatic creation will be performed and you can get the path as a string with either the `getPath` or `toString` methods on the `Path` object.

7.8 Multiple Return Values

D-Bus also allows functions to return multiple values, a concept not supported by Java. This has been solved in a fashion similar to the struct, using a `Tuple`[17] class. Tuples are defined as generic tuples which can be parameterised for different types and just need to be defined of the appropriate length. This can be seen in figure 13 and a call in figure 14. Again, these can be automatically generated from introspection data.

```
import org.freedesktop.dbus.Tuple;

public final class TestTuple<A, B, C> extends Tuple
{
   public final A a;
   public final B b;
   public final C c;
   public TestTuple(A a, B b, C c)
   {
      this.a = a;
      this.b = b;
      this.c = c;
   }
}
```

Figure 13: A 3-tuple

```
public ThreeTuple<String, Integer, Boolean> status(int item);
```

Figure 14: A Tuple being returned from a method

7.9 Full list of types

Table 1 contains a full list of all the Java types and their corresponding D-Bus types.

[17] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/Tuple.html`

Java Type	**D-Bus Type**
Byte	DBUS_TYPE_BYTE
byte	DBUS_TYPE_BYTE
Boolean	DBUS_TYPE_BOOLEAN
boolean	DBUS_TYPE_BOOLEAN
Short	DBUS_TYPE_INT16
short	DBUS_TYPE_INT16
UInt16	DBUS_TYPE_UINT16
int	DBUS_TYPE_INT32
Integer	DBUS_TYPE_INT32
UInt32	DBUS_TYPE_UINT32
long	DBUS_TYPE_INT64
Long	DBUS_TYPE_INT64
UInt64	DBUS_TYPE_UINT64
double	DBUS_TYPE_DOUBLE
Double	DBUS_TYPE_DOUBLE
String	DBUS_TYPE_STRING
Path	DBUS_TYPE_OBJECT_PATH
<T>	DBUS_TYPE_VARIANT
Variant	DBUS_TYPE_VARIANT
? extends Struct	DBUS_TYPE_STRUCT
?[ ]	DBUS_TYPE_ARRAY
? extends List	DBUS_TYPE_ARRAY
? extends Map	DBUS_TYPE_DICT
? extends DBusInterface	DBUS_TYPE_OBJECT_PATH
Type[ ]	DBUS_TYPE_SIGNATURE

Table 1: Mapping between Java types and D-Bus types

7.9.1 float

Currently the D-Bus reference implementation does not support a native single-precision floating point type. Along with the C# implementation of the protocol, the Java implementation supports this extension to the protocol. By default, however, the library operates in compatibility mode and converts all floats to the double type. To disable compatibility mode export the environment variable `DBUS_JAVA_FLOATS=true`.

8 Annotations

You can annotate your D-Bus methods as in figure 15 to provide hints to other users of your API. Common annotations are listed in table 2.

```
package org.freedesktop;
import org.freedesktop.dbus.UInt32;
import org.freedesktop.dbus.DBusInterface;

@org.freedesktop.DBus.Description("Some Methods");
public interface DBus extends DBusInterface
{
   @org.freedesktop.DBus.Description("Check if the name has an owner")
   public boolean NameHasOwner(String name);
   @org.freedesktop.DBus.Description("Request a name")
   @org.freedesktop.DBus.Deprecated()
   public UInt32 RequestName(String name, UInt32 flags);
}
```

Figure 15: An annotated method

Name	Meaning
org.freedesktop.DBus.Description	Provide a short 1-line description of the method or interface.
org.freedesktop.DBus.Deprecated	This method or interface is Deprecated.
org.freedesktop.DBus.Method.NoReply	This method may be called and returned without waiting for a reply.
org.freedesktop.DBus.Method.Error	This method may throw the listed Exception in addition to the standard ones.

Table 2: Common Annotations

9 DBusSerializable

Some people may want to be able to pass their own objects over D-Bus. Obviously only raw D-Bus types can be sent on the message bus itself, but the Java implementation

allows the creation of serializable objects which can be passed to D-Bus functions and will be converted to/from D-Bus types by the library.

To create such a class you must implement the `DBusSerializable`[18] class and provide two methods and a zero-argument constructor. The first method has the signature `public Object serialize() throws DBusException` and the second must be called `deserialize`, return `null` and take as it's arguments exactly all the dbus types that are being serialized to *in order* and *with parameterization*. The serialize method should return the class properties you wish to serialize, correctly formatted for the wire (`DBusConnection.convertParameters()` can help with this), in order in an Object array.

An example of a serializable class can be seen in figure 16.

10 CreateInterface

D-Bus provides a method to get introspection data on a remote object, which describes the interfaces, methods and signals it provides. This introspection data is in XML format[19]. The library automatically provides XML introspection data on all objects which are exported by it. Introspection data can be used to create Java interface definitions automatically.

The `CreateInterface`[20] class will automatically create Java source files from an XML file containing the introspection data, or by querying the remote object over D-Bus. CreateInterface can be called from Java code, or can be run as a stand alone program.

The syntax for the CreateInterface program is

```
CreateInterface [--system] [--session] [--create-files]
                  <bus name> <object>
CreateInterface [--create-files] <introspection-file.xml>
```

The Java source code interfaces will be written to the standard ouput. If the `--create-files` option is specified the correct files in the correct directory structure will be created.

10.1 Nested Interfaces

In some cases there are nested interfaces. In this case CreateInterface will not correctly create the Java equivalent. This is because Java cannot have both a class and a package with the same name. The solution to this is to create nested classes in the same file.

An example would be the Hal interface:

```
<interface name="org.freedesktop.Hal.Device">
   ...
```

[18] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DBusSerializable.html`

[19] http://dbus.freedesktop.org/doc/dbus-specification.html#introspection-format

[20] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/CreateInterface.html`

```
</interface>
<interface name="org.freedesktop.Hal.Device.Volume">
   ...
</interface>
```

When converted to Java you would just have one file `org/freedesktop/Hal/Device.java` in the package `org.freedesktop.Hal`, which would contain one class and one nested class:

```
public interface Device extends DBusInterface {
   public interface Volume extends DBusInterface {
      ... methods in Volume ...
   }
   ... methods in Device ...
}
```

11 Debugging

It is possible to enable debugging in the library. This will be a lot slower, but can print a lot of useful information for debugging your program.

To enable a debug build compile with `DEBUG=enable`. This will then need to be enabled at runtime by using the debug jar with debugging enabled (usually installed as debug-enable.jar alongside the normal jar).

Running a program which uses this library will print some informative messages. More verbose debug information can be got by supplying a custom debug configuration file. This should be placed in the file `debug.conf` and has the format:

```
classname = LEVEL
```

Where `classname` is either the special word `ALL` or a full class name like `org.freedesktop.dbus` and `LEVEL` is one of `NONE`, `CRIT`, `ERR`, `WARN`, `INFO`, `DEBUG`, `VERBOSE`, `YES`, `ALL` or `TRUE`. This will set the debug level for a particular class. Any messages from that class at that level or higher will be printed. Verbose debugging is extremely verbose.

In addition, setting the environment variable `DBUS_JAVA_EXCEPTION_DEBUG` will cause all exceptions which are handled internally to have their stack trace printed when they are handled. This will happen unless debugging has been disabled for that class.

12 Peer to Peer

It is possible to connect two applications together directly without using a bus. D-Bus calls this a peer-to-peer connection.

The Java implementation provides an alternative to the `DBusConnection` class, the `DirectConnection`[21] class. This allows you to connect two applications together directly without the need of a bus.

[21] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/DirectConnection.html`

When using a DirectConnection rather than a DBusConnection most operations are the same. The only things which differ are how you connect and the operations which depend on a bus. Since peer connections are only one-to-one there is no destination or source address to messages. There is also no `org.freedesktop.DBus` service running on the bus.

12.1 Connecting to another application

To connect with a peer connection one of the two applications must be listening on the socket and the other connecting. Both of these use the same method to instantiate the `DirectConnection` but with different addresses. To listen rather than connect you add the *"listen=true"* parameter to the address. Listening and connecting can be seen in figures 17 and 18 respectively. Listening will block at creating the connection until the other application has connected.

`DirectConnection` also provides a `createDynamicSession` method which generates a random abstract unix socket address to use.

12.2 Getting Remote Objects

Getting a remote object is essentially the same as with a bus connection, except that you do not have to specify a bus name, only an object path. There is also no `getPeerRemoteObject` method, since there can only be one peer on this connection.

The rest of the API is the same for peer connections as bus connections, ommiting any bus addresses.

13 Low-level API

In very rare circumstances it may be neccessary to deal directly with messages on the bus, rather than with objects and method calls. This implementation gives the programmer access to this low-level API but its use is strongly recommended against.

To use the low-level API you use a different set of classes than with the normal API.

13.1 Transport

The `Transport`[22] class is used to connect to the underlying transport with a bus address and to send and receive messages.

You connect by either creating a `Transport` object with the bus address as the parameter, or by calling `connect` with the address later. Addresses are represented using the `BusAddress` class.

Messages can be read by calling `transport.min.readMessage()` and written by using the `transport.mout.writeMessage(m)` methods.

[22] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/Transport.html`

13.2 Message

`Message`[23] is the superclass of all the classes representing a message. To send a message you need to create a subclass of this object. Possible message types are: `MethodCall`, `MethodReturn`, `Error` and `DBusSignal`. Constructors for these vary, but they are basically similar to the `MethodCall` class.

All the constructors have variadic parameter lists with the last of the parameters being the signature of the message body and the parameters which make up the body. If the message has an empty body then the last parameter must be null. Reading and writing messages is not thread safe.

Messages can be read either in blocking or non-blocking mode. When reading a message in non-blocking mode, if a full message has not yet been read from the transport the method will return null. Messages are instantiated as the correct message type, so `instanceof` will work on the returned object. Blocking mode can be enabled with an extra parameter to the Transport constructor.

Figure 20 shows how to connect to a bus, send the (required) initial 'Hello' message and call a method with two parameters.

14 Examples

As an example here are a complete set of interfaces for the bluemon[24] daemon, which communicates over D-Bus. These interfaces were all created by querying introspection data over the bus.

[23] `http://dbus.freedesktop.org/doc/dbus-java/api/org/freedesktop/dbus/Message.html`
[24] http://www.matthew.ath.cx/projects/bluemon

15 Credits

This document and the Java API were written by and are copyright to Matthew Johnson. Much help and advice was provided by the members of the #dbus irc channel. Comments, corrections and patches can be sent to dbus-java@matthew.ath.cx.

```
import java.lang.reflect.Type;

import java.util.List;
import java.util.Vector;

import org.freedesktop.dbus.DBusConnection;
import org.freedesktop.dbus.DBusSerializable;

public class TestSerializable implements DBusSerializable
{
   private int a;
   private String b;
   private Vector<Integer> c;
   public TestSerializable(int a, String b, Vector<Integer> c)
   {
      this.a = a;
      this.b = b.toString();
      this.c = c;
   }
   public TestSerializable() {}
   public void deserialize(int a, String b, List<Integer> c)
   {
      this.a = a;
      this.b = b;
      this.c = new Vector<Integer>(c);
   }
   public Object[] serialize()
   {
      return new Object[] { a, b, c };
   }
   public int getInt() { return a; }
   public String getString() { return b; }
   public Vector<Integer> getVector() { return c; }
   public String toString()
   {
      return "TestSerializable{"+a+","+b+","+c+"}";
   }
}
```

Figure 16: A serializable class

```
DirectConnection dc = new DirectConnection("unix:path=/tmp/dbus-ABCXYZ,listen=true");
```

Figure 17: Listening for a peer connection

```
DirectConnection dc = new DirectConnection("unix:path=/tmp/dbus-ABCXYZ");
```

Figure 18: Connecting to a peer connection

```
RemoteInterface remote = dc.getRemoteObject("/Path");
remote.doStuff();
```

Figure 19: Getting a remote object on a peer connection

```
BusAddress address = new BusAddress(
         System.getenv("DBUS_SESSION_BUS_ADDRESS"));
Transport conn = new Transport(address, true);

Message m = new MethodCall("org.freedesktop.DBus", "/org/freedesktop/DBus",
                           "org.freedesktop. DBus", "Hello", (byte) 0, null);
conn.mout.writeMessage(m);

m = conn.min.readMessage();
System.out.println("Response to Hello is: "+m);

m = new MethodCall("org.freedesktop.DBus", "/org/freedesktop/DBus",
                   "org.freedesktop.DBus", "RequestName", (byte) 0,
                   "su", "org.testname", 0);
conn.mout.writeMessage(m);

conn.disconnect();
```

Figure 20: Low-level usage

```
package cx.ath.matthew.bluemon;
import org.freedesktop.dbus.DBusInterface;
import org.freedesktop.dbus.UInt32;
public interface Bluemon extends DBusInterface
{
  public Triplet<String, Boolean, UInt32>
  Status(String address);
}
```

Figure 21: cx/ath/matthew/bluemon/Bluemon.java

```
package cx.ath.matthew.bluemon;
import org.freedesktop.dbus.DBusInterface;
import org.freedesktop.dbus.DBusSignal;
import org.freedesktop.dbus.exceptions.DBusException;
public interface ProximitySignal extends DBusInterface
{
   public static class Connect extends DBusSignal
   {
      public final String address;
      public Connect(String path, String address)
                                 throws DBusException
      {
         super(path, address);
         this.address = address;
      }
   }
   public static class Disconnect extends DBusSignal
   {
      public final String address;
      public Disconnect(String path, String address)
                                 throws DBusException
      {
         super(path, address);
         this.address = address;
      }
   }
}
```

Figure 22: cx/ath/matthew/bluemon/ProximitySignal.java

```
package cx.ath.matthew.bluemon;
import org.freedesktop.dbus.Tuple;
/** Just a typed container class */
public final class Triplet <A,B,C> extends Tuple
{
   public final A a;
   public final B b;
   public final C c;
   public Triplet(A a, B b, C c)
   {
      super(a, b, c);
      this.a = a;
      this.b = b;
      this.c = c;
   }
}
```

Figure 23: cx/ath/matthew/bluemon/Triplet.java

```
package cx.ath.matthew.bluemon;
import org.freedesktop.dbus.DBusConnection;
import org.freedesktop.dbus.DBusSigHandler;
import org.freedesktop.dbus.DBusSignal;
import org.freedesktop.dbus.UInt32;
import org.freedesktop.dbus.exceptions.DBusException;

public class Query {
   public static void main(String[] args) {
      String btid;
      Triplet<String, Boolean, UInt32> rv = null;

      if (0 == args.length) btid = "";
      else btid = args[0];

      DBusConnection conn = null;
      try {
         conn = DBusConnection.getConnection(DBusConnection.SYSTEM);
      } catch (DBusException De) {
         System.exit(1);
      }
      Bluemon b = (Bluemon) conn.getRemoteObject(
            "cx.ath.matthew.bluemon.server",
            "/cx/ath/matthew/bluemon/Bluemon", Bluemon.class);
      try {
         rv = b.Status(btid);
      } catch (RuntimeException Re) {
         System.exit(1);
      }
      String address = rv.a;
      boolean status = rv.b;
      int level = rv.c.intValue();

      if (status)
         System.out.println("Device "+address+
                            " connected with level "+level);
      else
         System.out.println("Device "+address+" not connected");
      conn.disconnect();
   }
}
```

Figure 24: cx/ath/matthew/bluemon/Query.java

```
/* cx/ath/matthew/bluemon/Client.java */
package cx.ath.matthew.bluemon;

import org.freedesktop.dbus.DBusConnection;
import org.freedesktop.dbus.DBusSigHandler;
import org.freedesktop.dbus.DBusSignal;
import org.freedesktop.dbus.exceptions.DBusException;

public class Client implements DBusSigHandler
{
   public void handle(DBusSignal s)
   {
      if (s instanceof ProximitySignal.Connect)
         System.out.println("Got a connect for "
               +((ProximitySignal.Connect) s).address);
      else if (s instanceof ProximitySignal.Disconnect)
         System.out.println("Got a disconnect for "
               +((ProximitySignal.Disconnect) s).address);
   }
   public static void main(String[] args)
   {
      System.out.println("Creating Connection");
      DBusConnection conn = null;
      try {
         conn = DBusConnection
                  .getConnection(DBusConnection.SYSTEM);
      } catch (DBusException DBe) {
         System.out.println("Could not connect to bus");
         System.exit(1);
      }

      try {
         conn.addSigHandler(ProximitySignal.Connect.class,
         new Client());
         conn.addSigHandler(ProximitySignal.Disconnect.class,
         new Client());
      } catch (DBusException DBe) {
         conn.disconnect();
         System.exit(1);
      }
   }
}
```

Figure 25: cx/ath/matthew/bluemon/Client.java