Scripting the Linux D-Bus with ooRexx

Is Projektseminar SS 2011

Vienna University of Economics and Business Administration

Advisor: Prof. Mag. Dr. Rony G. Flatscher

Author: BSc. Sebastian Margiol

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Sebastian Margiol

Wien, am 21. Juni 2011

Table of Contents

1 Introducing the D-Bus....5
1.1 History....5
1.2 Concept....6
1.2.1 Message Bus....8
1.2.2 Object Types....8
1.2.3 Message Types....9
1.2.4 Object Paths....9
1.2.5 Interfaces....9
1.2.6 Bus Names....10
1.2.7 Member Names....10
1.3 Spreading....10
1.4 Bindings....10
1.4.1 Java Binding....11
1.4.2 Python Binding....12

2 Investigating the Dbus....13
2.1 D-Feet....13
2.2 qdbusviewer....14
2.3 DbusViewer....15
2.4 Bustle....16
2.5 Common Interfaces....17

3 Implementation....19
3.1 Setting up the Environment....19
3.2 Connection to the D-Bus....20
3.3 Get Connection to a Remote Object....21
3.4 Controlling a Media Player....21
3.5 Controlling a Text Editor....26
3.6 Request a Bus Name....30
3.7 Controlling Skype....31
3.8 Troubleshooting....37

4 Roundup and Outlook....39

5 Bibliography.....40

6 Appendix.....42

Table of Figures

Figure 1: D-Bus overview diagram.....7

Figure 2: D-Feet DBus debugger.....14

Figure 3: qdbusviewer DBus debugger (qt4-dev-tools).....15

Figure 4: DbusViewer (java-dbus package).....16

Figure 5: Bustle D-Bus profiler output.....17

Figure 6: Setting the Classpath 20

Figure 7: Getting connected to the D-Bus.....20

Figure 8: D-Feet information about vlc program.....22

Figure 9: Creation and compilation of necessary Java Files (vlc1.rxj).....22

Figure 10: Invoke actions of a mediaplayer program (vlc2.rxj).....23

Figure 11: Convert return value from mediaplayer.....24

Figure 12: Handle a Dbus-Map (vlc3.rxj).....24

Figure 13: Connection to a signal (vlc_signal.rxj).....25

Figure 14: Create and compile Java Files for a multi instance application.....26

Figure 15: Invoke methods on a texteditor (kate2.rxj).....27

Figure 16: Parameters for klipper program.....28

Figure 17: Interaction between kate and klipper (kate3.rxj).....29

Figure 18: Change content of texteditor document (kate4.rxj).....30

Figure 19: Export objects via D-Bus (exportObject.rxj).....31

Figure 20: Skype configuration file (/etc/dbus-1/system.d/skype.conf).....32

Figure 21: Skype access confirmation dialog.....33

Figure 22: Accessing the Skype API (skype1.rxj).....34

Figure 23: Effect a test call with Skype (skype2.rxj).....34

Figure 24: Create a chat with Skype (skype3.rxj).....35

Figure 25: Skype commands.....35

Figure 26: Send birthday wishes automatically (skype_birthday.rxj).....37

Figure 27: Rexxs excellent String capabilities.....37

Figure 28: Appendix: Test.py.....42

Figure 29: Appendix: Test.java.....42

Figure 30: Appendix: Struct3.java.....43

Figure 31: Appendix: Document.java.....43

Figure 32: Sample script D-Bus control gui (app.rxj).....48

Abstract

This paper gives a short introduction to the D-Bus on Linux operating systems. D-Bus offers among other things an InterProcessCommunication (IPC) mechanism. The programming language ooRexx uses the Bean Scripting Framework BSF4ooRexx and the Java binding to access the D-Bus. A binding offers support for the given programming language, enabling communication with any connected program on the D-Bus, no matter in what language it was implemented. The sample scripts in the implementation chapter are demonstrating the necessary steps to interact with applications like VLC MediaPlayer, Kate, Klipper and Skype over the D-Bus. That means invoke methods, process return values and listening for signals. D-Bus offers an interesting possibility for an ooRexx programmer to orchestrate applications in order to realize automation of work-flow.

Abstrakt

Diese Arbeit liefert eine kurze Einführung in das D-Bus Konzept unter Linux Betriebssystemen. D-Bus bietet unter anderem die Funktionalität eines InterProcessCommunication (IPC) Mechanismus. In der Programmiersprache ooRexx wird über das Bean Scripting Framework BSF4ooRexx die Java Anbindung des D-Bus verwendet. Eine Anbindung hat zum Ziel die D-Bus Funktionalität für die jeweilige Programmiersprache zur Verfügung zu stellen, wodurch Programme mit anderen verbundenen Programm interagieren können, egal in welcher Programmiersprache sie implementiert wurden. Die Applikationen VLC MediaPlayer, Kate, Klipper und Skype werden in den Beispielskripten eingesetzt um die notwendige Schritte zu demonstrieren um über den D-Bus zu interagieren. Das bedeutet bestimmte Funktionen eines Programms aufzurufen, Rückgabewerte zu verarbeiten und auf Signale zu warten. Einem ooRexx Programmierer wird mit D-Bus eine interessante Möglichkeit geboten, um über eine geschickte Orchestrierung von Anwendungen, Arbeitsabläufe zu automatisieren.

1 Introducing the D-Bus

The goal of the D-Bus is to provide an unified mechanism to exchange messages across the bus among connected applications, no matter what program language they are implemented in. Via a D-Bus connection, every application can provide services which can get accessed from another application. Transported object types get marshalled from their native implementation to the bus specification, routed through the bus to a specified receiver and get demarshalled again. This ensures interoperability. The marshalling is done by the language binding to the D-Bus. The D-Bus got its name from the central server application that operates as a daemon, called bus, which handles the communication. [Tro11]

1.1 History

The list of mentioned predecessor is by far not complete. Arexx was chosen because its affinity to ooRexx[1], although it does not really represent an IPC system and DCOP was chosen because KDE is largely utilized on desktop computers and a competition of D-Bus and DCOP would destroy the idea of one unified IPC mechanism for all Linux systems (or for all operating systems at all).

Arexx

On the Amiga Platform, the programming language Arexx emerged as a linking knot for different applications. Arexx was used for operations like multitasking and inter process communication. The latter was realized through static named message ports (so called RexxPorts), over which applications can send and receive messages. During the initialization of an application, a message port is opened and the application waits for incoming messages. When a message arrives the port, the operating systems notifies the receiver, which processes the message and sends back a response to the Arexx program. The Arexx program itself can forward the message to other programs in order to instruct them do proceed with results. [Deb02] So Arexx represents an mechanism for sys-

[1] And because of my sympathy for the Amiga platform.

tem-wide interprocess communication, which connects applications and orchestrate their interplay, given they offer RexxPorts.

DCOP

The Desktop COmmunication Protocol (DCOP) was developed to link all KDE[2] applications together. The DCOP server dispatches messages to connected applications. This message could either be a *'send and forget'* message (non-blocking), or a *'call'* which waits for answers. Signals are supported as well. Transported messages get serialized and retranslated. [Kde07] Since Version 4, KDE dropped DCOP in favor of D-Bus. This is a very welcome development as a competitor of D-Bus disappears.

On the GNOME[3] desktop environment a CORBA-like component model handled IPC mechanisms but was also replaced by D-Bus. [Fre11c]

1.2 Concept

D-Bus launches a central server which keeps track of all connected applications and handles the message routing among them. Connections can be established to the session bus and to the system bus. The connection is effected through a socket which allows bidirectional message exchange. Each Socket has a unique address. Instances of applications can be addressed with an object path, each object announce its services through interfaces. Applications are communicating over the D-Bus through messages and signals.

D-Bus is a low-latency system for interprocess communication, as it avoids round trips and allows asynchronous operations[4]. It has a low overhead as it uses a binary protocol, and it is easy to deploy as messages are exchanged rather than byte streams. [Fre11b] Many bindings for programming languages are provided.

The main feature provided through the protocol is the D-Bus message bus, which handles connections from applications and exchanges messages among

[2] http://www.kde.org/
[3] Http://www.gnome.org/
[4] If a call is effected synchronous, any other incoming message will get queued. This could lead to time inefficiency.

them. The D-Bus is used for notifications of system changes[5] and desktop application interoperability[6].

Figure 1 represents an interaction of two applications via the D-Bus.

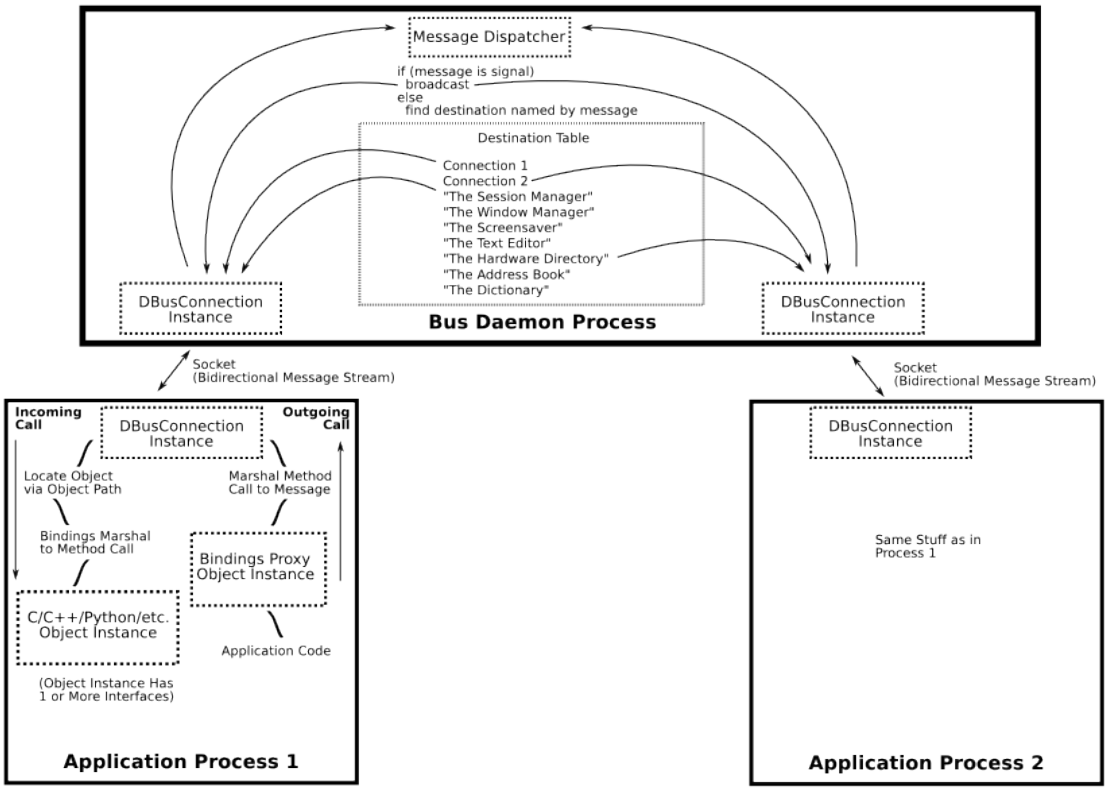

Figure 1: D-Bus overview diagram. Source: http://dbus.freedesktop.org/doc/diagram.png

The Bus Daemon[7] Process is started in an early stage of the system's init process. An application connects to the D-Bus through a socket, installing a bidirectional message stream through which messages can be forwarded. Every application demanding a connection to the D-Bus is registered within the bus and can be identified with a unique number or name.

An important condition for a successful interaction of applications is the programming language Independence. Therefore help of a translation mechanism is needed. This can be achieved by using a binding which translates the objects of an application into a (neutral) D-Bus representation[8]. This mechanism

[5] When an usb device is attached, a signal is broadcasted through the bus. All applications that might be interested in such an event are listening for that signal.

[6] When a VOIP telephone call arrives, the mediaplayer could get paused and restartet after the call is finished or refused.

[7] For further information look at http://dbus.freedesktop.org/doc/dbus-daemon.1.html

[8] Examples for D-Bus object type representations can be seen in chapter 1.2.2

is called marshalling. When an application written in Java wants to exchange information with another application, the information is marshalled through its binding and forwarded to the D-Bus daemon[9]. The daemon checks what kind of message the application sends. If it is a signal, it is broadcasted over the D-Bus and every application, registered for this very signal gets informed. If the message is an information for a specific application, the daemon looks up its registry whether the desired bus address is registered and forwards the message to it. On this address the target object is identified by its path name and the forwarded message gets marshalled again to the application's native representation. The next sub-chapters are covering the mentioned terms.

1.2.1 Message Bus

The message bus provides features like single-owner bus names, on-demand startup of services, and security policies. [Fre11b]

The message bus is divided into a system bus and a session bus. The system bus routes messages from the system to user sessions and/or requests input from them. A system event, like the network status is broadcasted via the system bus. As there are restrictions about what services an application may offer on this bus, the services on this bus are likely be offered by a system known application. [Tro11]

On the session bus, desktop environments are implemented. [Fre11b] Installed software that is not part of the operating system usually connects to the session bus.

1.2.2 Object Types

Basic object types are supported by the D-Bus (*Byte, Boolean,* different types of *Integers, Doubles* and *Strings*) as well as four container types, namely *Struct, Array, Variant and Dict_entry*. D-Bus provides its own marshalling, [Fre11a] the types are not identified by tags in their marshalled data, but with a type code. This type code is an ASCII character representation of the value type. For example a single 32-bit *Integer* is denoted as *'i'*, a string as *'s'* and a

[9] Messages can also be exchanged on low-level, binding are not absolutely necessary for that purpose.

boolean as *'b'*. Structs are identified by *'( )'*, for example a Struct with a String and an Integer is processed as *(si)*. [Fre11b]

1.2.3 Message Types

D-Bus supports four message types, *METHOD_CALL*, *METHOD_RETURN*, *ERROR* and *SIGNAL*. *METHOD_CALL* invokes operations on remote objects, which replies with either *METHOD_RETURN* or *ERROR*. [Fre11b]

SIGNAL messages do not expect a reply, they are used for information distribution. Signals are single, unidirectional messages without any specified receiver. Their information is broadcasted on the Bus. Client applications which register for a particular signal, get informed as soon as it is emitted. It does not matter how many clients receive a copy of the signal. [Fre11b] The Vlc media player for example broadcasts signals about about status changes (such as pause) or music track changes.

1.2.4 Object Paths

The object path is the name of an objects instance (communications endpoint). It is called path because it looks exactly like a path in a file system hierarchy. If an application wants to expose more objects, they are distinguished by their path. The application vlc for example offers */Player* and */TrackList*, each having their own interface.

A valid object path must follow these constraints: begin with '/' and contain [A-Z][a-z][0-9]_ characters. Empty values and multiple '/' are not allowed. [Fre11b]

1.2.5 Interfaces

All methods that are made available through the D-Bus have to be specified in interfaces. Interfaces are sets of declarations, like a dictionary which informs about what methods are provided and how their signature looks.

The name of an interface has to be separated by a '.', but must not begin with it and each separated element must contain the characters [A-Z][a-z][0-9]_. For example *'org.mpris.mediaplayer2'.* [Fre11b]

1.2.6 Bus Names

Bus names are the names of connections. Applications can either be called by their unique name representation (for example :35-731) or with a name (for example org.kde.KTextEditor). A program can request any well known name to be identifiable easily, if the chosen name is not already registered with an other program. If an application allows multiple instances and gets nonetheless be called with a well known name, a process number is added to the name to distinguish (for example org.kde.kate-1421).

1.2.7 Member Names

Within an interface, all methods and signals are exposed with a name. This names are also called members of the interface. D-Bus does not support method overloading, thus cannot distinguish methods on the basis of their object types. Therefore each method and signal must have a unique name. [Fre11b]

1.3 Spreading

The D-Bus system is part of any actual Gnome and KDE Desktop, so it is already available for many Linux users, and it is easily portable to any Linux distribution. There are efforts to port it to the Windows Platform, as well. [Fre11c]

A free standard that is used within many systems is very attractive for software developers as they can easily develop compliant applications. The more programs connected to the D-Bus, the more interesting it could be to connect an own application. This network effect could D-Bus help to have a promising future.

1.4 Bindings

Generally spoken, a binding offers access to an underlying architecture, in an easier and more comprehensive way. The goal of a D-Bus binding is to map the D-Bus API to the a personally preferred programming language as naturally as possible. [Fre11a]

The low-level implementation of the D-Bus system serves the basis for bindings implementations. The only dependency for the low-level libdbus reference implementation is an XML parser (libxml or expat). The libdbus implementation is designed especially for binding authors and represent a reference for reimplementation.[10] For programmers of application using the D-Bus, it is recommended to use a higher level language binding instead of a low-level access. [Fre11b]

There are also bindings available that were not built on top of the libdbus reference implementation, thus not representing a wrapper around the low-level API, but a reimplementation, for example the C#, Java and Ruby binding. [Fre11b]

There are plenty of bindings available for different programming languages, for a complete list look at http://www.freedesktop.org/wiki/Software/DBusBindings.

These Bindings do not only differ in their program language vocabulary, but also in their ease of use, especially in their dependencies on libraries and the amount of necessary code needed to interact through the D-Bus.

In spite of the lack of a direct D-Bus-Binding for ooRexx[11], the connection to the Bus with ooRexx can be achieved through an indirect way via BSF4ooRexx and the Java binding respectively.

1.4.1 Java Binding

Java users have the choice between different versions of D-Bus access. All versions since 2.0 are complete native reimplementations which do not wrap the libdbus reference implementation like 1.x versions do. [Fre11a] The Java approach to the D-Bus is a little exhausting as a lot of Java classes have to be created in order to achieve connection. If a method on a remote object is to be called, a Java Interface has to be implemented as well as classes for each parameter or return value, if they are non standard Java objects. [Joh11a]

[10] Information about the low-level D-Bus C API can be found at http://dbus.freedesktop.org/doc/api/html/.
[11] What is very sad, considering the role Arexx had once.

This binding will be seen in action in the implementation chapter with ooRexx examples.[12]

1.4.2 Python Binding

In this paper this binding will not be used. The reason why python-dbus is mentioned is that there are many bindings available and some of them might be easier of use than others, even if a programmer is not that familiar with the language.

The D-Bus is statically typed, that means that the argument types of method invocations must exactly match the signature of the call. Arguments of incorrect types will be ignored and thus the method call will not be effected. The introspection mechanism of D-Bus offers the possibility to identify the correct argument type. The python binding uses this mechanism in order to convert native Python types automatically[13].

What this means in terms of the easiness to use and the necessary amount of Lines of Code is demonstrated in the Appendix.

[12] As the application '*CreateInterface'* will be used instead of creating every class by class, it might be useful to read this document for better understanding.
http://dbus.freedesktop.org/doc/dbus-java/dbus-java.pdf

[13] For a full list of supported types refer to basic types:
http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html

2 Investigating the Dbus

As mentioned in the previous chapters, every application uses its own, unique bus name and gets identified at this address with its object path. Communication through the bus can be monitored issuing the command dbusmonitor in a shell. Every single message is listed with information about the sender and the receiver. Signals are listed with their sender only, as they do not have specific receivers. As every single message or signal from every application is listed, this tool is not very helpful to inspect applications due to the amount of information.[14]

There are excellent applications for investigation purpose, listening all connected programs within a functional and comprehensive graphical user interface, for example the D-Bus debuggers d-feet or qdbusviewer.

If a program that is expected to provide D-Bus support is not listed, you might check if its D-Bus connectivity is activated by default[15] or it has to be activated manually. If there is no such option, it is unlikely that the program offers D-Bus support.

2.1 D-Feet

The program D-Feet[16] is available in the standard repository of many Debian based systems. Is written in python and therefore uses the dbus-python binding.

On the left side of the panel, all connected applications are listed, on selecting one of them, additional information about provided objects (object paths), the interface names as well as all methods and signals declared through it are displayed.

[14] Filters can be used to monitor specified applications, but that premises that you already know something about this application.
[15] For example VLC offers plenty of possibilities to control the mediaplayer, but none of them by default.
[16] http://live.gnome.org/DFeet/

Figure 2: D-Feet DBus debugger

On the left side of the panel, all connected applications are listed, on selecting one of them, additional information about the provided objects (object paths), the interface names as well as all methods and signals declared with it are displayed.

A very handy feature is to hide private Bus-addresses. As it is not possible to effect a public connection to them, the list is much clearer arranged without them being listed. All methods of the interfaces are listed with information about necessary parameters and which object type the parameters are of. The method *endOfLine* (see figure) needs an integer value for the line, and returns a Struct (data container) with two integers. It is very easy and useful to check the signature of methods with this program. Method calls can be tested right within this program. But the parameters are coded in python, so a successful method invocation in D-Feet, might need another syntax in a different programming language.

2.2 qdbusviewer

Another very nice application in this area is called qdbusviewer[17] which is packed in the qt4-dev-tools. The structure is quite similar to D-Feet's, on the left side all connected applications are listed, also the private ones, it is not possible to hide them. They are identifiable as they are listed with a bus num-

[17] http://doc.qt.nokia.com/4.5/qdbusviewer.html

ber and not with a name. A misleaded click could freeze the application for a few seconds as it attempts to connect.

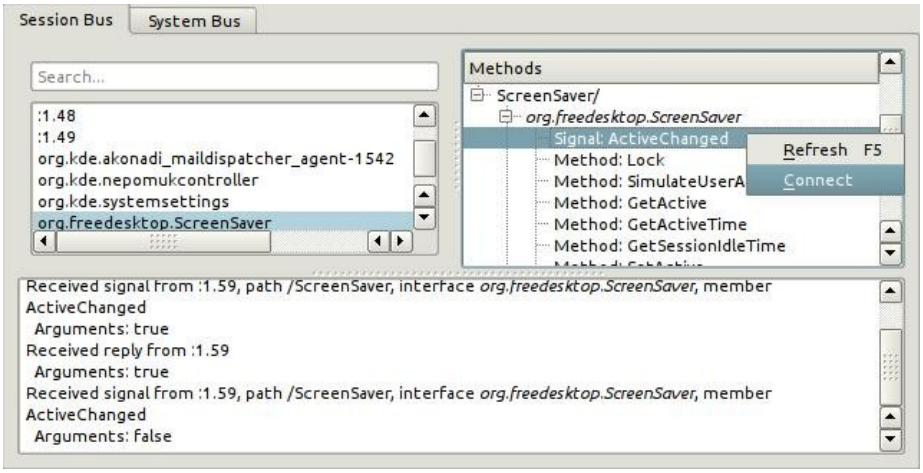

Figure 3: qdbusviewer DBus debugger (qt4-dev-tools)

There is a very useful feature which is not available in D-feet. It is possible to connect to signals of an interface. Figure 3 demonstrate a signal connection to the signal *ActiveChanged* of the interface org.freedesktop.ScreenSaver. Whenever the screensaver starts or stops, its activechanged signal is sent through the dbus and applications that have claimed an interest on this signal get informed.

If a method is called, a pop-up dialog supplies information about the needed parameters and their object type, but there is no information about the object type and the structure of return values from an application.

2.3 DbusViewer

Another D-Bus debugger, shipped with the java-dbus package is called DbusViewer and can be activated via the command line. This tool inspects every connection at startup, even the private ones, which then rejects the connection attempt. Therefore it takes a long time until DbusViewer is ready for usage.

Figure 4: DbusViewer (java-dbus package)

Figure 4 shows the inspection information of an application called kate. All necessary classes for the usage via dbus-java are created via the introspection data of the interface. For patient users,[18] this might be useful, because desired Java classes can be extracted among all available classes. If only a specific call to a single method has to be effected, and the parameters or the return value are standard object types, only the interface has to be implemented and within that, only the desired method has to be declared. But as this topic goes beyond the investigation of the D-Bus, it will be covered in the implementation chapter.

2.4 Bustle

Bustle[19] is a D-Bus profiler witch draws up sequence diagrams of the message exchange through the bus.

[18] Referring to the startup time.
[19] http://willthompson.co.uk/bustle/

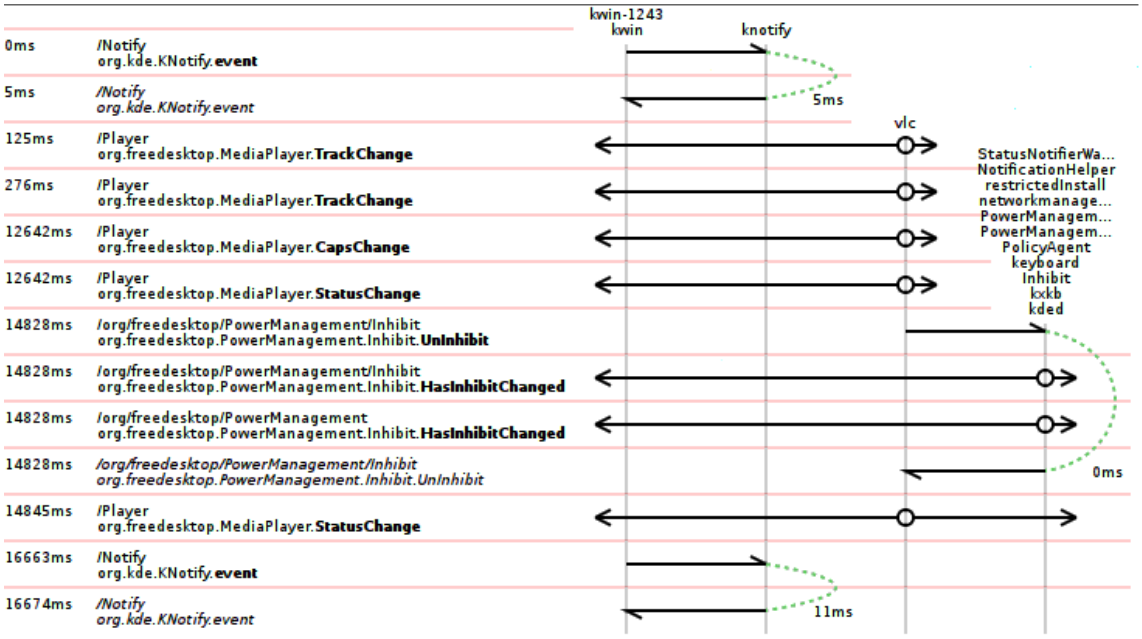

Figure 5: Bustle D-Bus profiler output

All exchanged messages and signals are listed with a time stamp of their invocation moment. This allows inspection on how active an application is on the D-Bus and how long it takes for messages to be sent and received. In this example a media player called vlc emits signals about its statuschange and trackchange. With the help of this profiler, information distribution bottlenecks can be identified and applications can be audited whether they emit signals or method calls which might be useless, thereby producing overhead.

2.5 Common Interfaces

It is not very handy to interact with similar applications doing common tasks, if they all have different interfaces. A generic interface for such applications is therefore very useful.

The Media Player Remote Interfacing Specification (MPRIS)[20] offers a standard for media players to execute standard playback functions (play, pause, stop, next, previous), media player state control as well as tracklist functions. Compliant players are Audacious, VLC, BMPx, QMMP, Dragon Player[21], Rhytmbox, Banshee and Amarok for example. This standard is available in the specifica-

[20] http://www.mpris.org
[21] http://incise.org/mpris-remote.html

tion 2.1 at the moment, but this version is not supported by all listed mediaplayers above.

If a media player wants to be compliant it has to request a unique bus name beginning with *org.mpris.MediaPlayer2*, the object path should be named */org/mpris/MediaPlayer2* and the interfaces *org.mpris.MediaPlayer2* and *org.mpris.MediaPlayer2.Player*. Connected clients (to signals) get informed about status changes of the media player via the *org.freedesktop.DBus.Properties.PropertiesChanged* signal.

A common interface for mediaplayers is useful as audio hardware keys could be mapped to D-Bus calls, and therefore reach any compliant mediaplayer. Some operating systems provide a widget which could control basic playback options for any mpris compliant mediaplayer. Thereby mediaplayers can be controlled from a central point. And it is useful for a programmer as well, as there is nearly no inspection necessary, except the specification of the used MPRIS interface.

For a summary and description of all available methods, properties and signals of the MPRIS interface specification 2.1, lookup the API at (http://www.mpris.org/2.1/spec/) or use a D-Bus debugger and inspect your favorite mediaplayer application on the bus. In the chapter 'implementation' of this paper, a connection to the vlc media player get established, which takes use of this common interface.[22] there are applications (Banshee for example) which are implementing their own interface, providing additional functions in addition to the MPRIS interface, but be nonetheless compliant to this standard.

[22] Vlc media player in version 1.1.9. implements the mpris standard 1.0. Information about the version 1.0 can be retrieved at (http://www.mpris.org/1.0/spec).

3 Implementation

In this chapter, the procedure for obtaining a connection to the D-Bus, invocation of methods, handling return values and exporting objects to the bus is explained with the help of a few examples. The goal of this chapter is to provide the essential knowledge for interaction with applications.

3.1 Setting up the Environment

If you do not already use a Linux operating system, you might take a look at the ooRexx homepage,[23] in the download section you will find information about releases, where supported operating systems are listed. If you do not know what distribution to choose, you might look up additional information about the Linux flavors on http://distrowatch.com/ and on the distributions homepage respectively.

Java is probably already installed, this can be tested by issuing *java -version* on the command line. If it is not installed, then make up for it through the distributions package manager.

Next, Open Object Rexx[24] has to be installed, as mentioned above you can obtain the package on the homepage. Then you need BSF4ooRexx,[25] this software allows ooRexx to use Java classes. With the help of BSF4ooRexx the java-dbus binding will be used to connect to the D-Bus. Due to the fact that the installation procedure uses ooRexx and tests the invocation of Java, you will get informed, should the installation of any dependency fail somehow. Next dbus-java has to be installed, either by searching it in your package manager (version 2.8 / Ubuntu 11.04) or download it from http://gitweb.freedesktop.org/?p=dbus/dbus-java.git (newest version available there is 2.7). With this package, three libraries are installed among others. In order to develop java-dbus applications, these libraries (dbus.jar, unix.jar and debug-enable.jar) must be made available. This can be effected by issuing following command in a terminal:

[23] http://www.oorexx.org/download.html
[24] http://www.oorexx.org/
[25] http://sourceforge.net/projects/bsf4oorexx/

```
export
CLASSPATH=$CLASSPATH::/usr/share/java/dbus.jar:/usr/share/java/unix.jar:/usr/share/java
/debug-enable.jar
```

Figure 6: Setting the Classpath

The command $CLASSPATH ensures that the values are appended and registered libraries are not overwritten. It might be necessary to adjust the path to the jar archives depending on the operating system. These *.jar files are links to the newest installed versions. Actually dbus.jar links to dbus-2.8.jar on my system. If you want to make the libraries permanent available, the .bashrc file has to be edited.

Optionally a dbus debugger (as described in chapter 2) can be installed. In these examples D-Feet is used to identify necessary parameters and test an method invocation on our ooRexx program.

3.2 Connection to the D-Bus

Now it is time to establish a connection to the D-Bus, this is very easy to achieve, there is only one dependency, a Java class called *'org.freedesktop.dbus.DBusConnection'*. This is a static and a singleton class, so it cannot be instanced and it is only possible to establish one connection to it. The same reference is returned when trying to get another connection.[26] Connections can be established to the system or session bus either. The static fields, representing the connection addresses are SYSTEM (value = 0) and SESSION (value = 1), but as only integer values are accepted in this case, the address has to be boxed.

```
-- import the Java class DBusConnection
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
-- establish a connection to the session bus
connection = .Connection~getConnection(box('int',.Connection~SESSION))
connection~disconnect
::requires BSF.CLS
```

Figure 7: Getting connected to the D-Bus

In this example a connection to the session bus is established and get closed again at the end of the script. To obtain connection to a remote object (applica-

[26] http://dbus.freedesktop.org/doc/dbus-java/api/

tion) Java classes have to be created. This will be demonstrated in the next chapter.

3.3 Get Connection to a Remote Object

For obtaining connection to a remote object it is necessary to implement a Java interface which denotes all available methods of this object. If the remote object expects object types that are not default ones[27], or distributes their return values in such types, a Java class representation has to be created for each of them.

This means a lot of programming effort for effecting an interaction with a remote object, therefore it is highly recommended to use the Java program *CreateInterface*, which was installed as part of the java-dbus package. This program creates all necessary Java files automatically. These files then have to be compiled in order to be ready to use for a ooRexx program. A language like ooRexx is ideally suited for executing the described tasks in a script.

3.4 Controlling a Media Player

In this section, control of my favorite media player, the VideoLAN Media Player (VLC)[28] will be executed.

In order to create all necessary classes and compile them, the bus name, the objectpath and the interface name[29] are important.

In the default settings VLC is not connected to the D-Bus. To activate the connection go to Settings and select *show expert mode*. The category interface listed on the left side contains a subcategory called control-interface; activate it and check the box named D-Bus control-interface. After a restart of the program it should be connected to the bus and can be inspected with a dbus-debugger.[30]

---

[27] See subchaper object types
[28] http://www.videolan.org/vlc/
[29] As the vlc media player complies to the mpris standard, its interface name is org.freedesktop.MediaPlayer
[30] D-Bus debugger have been described in chapter 2.

The program to inspect can be found through its name, the filtered list offers the bus address *'org.mpris.vlc'* what represents the desired application.

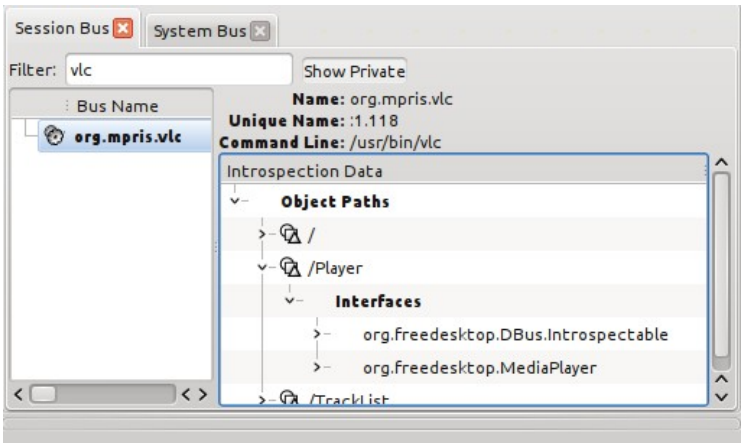

Figure 8: D-Feet information about vlc program

With information from D-Feet (Figure 8) about the program vlc on the session bus, the name *'org.mpris.vlc'* can be identified as well as three object paths. The path *'/Player'* lists the interface *'org.freedesktop.MediaPlayer'*, which offers interesting methods.

The bus name and the object path are parameters for the *CreateInterface* program and the interface name is needed for the compilation of the Java classes. Be aware of that the application to inspect has to be started before executing the CreateInterface program.

```
busname = 'org.mpris.vlc'
objectpath = '/Player'
interfacename = 'org.freedesktop.MediaPlayer'

-- create the interfaces with the given parameters
createinterface = 'CreateInterface -s -f' busname objectpath
createinterface

-- format the interfacename string to a path
tmp = TRANSLATE(interfacename, '/' , '.')
path = SUBSTR(tmp, 1, tmp~LASTPOS('/'))

-- compile the created Java classes
compile = 'javac ./'path'*.java'
compile
```

Figure 9: Creation and compilation of necessary Java Files (vlc1.rxj)

Java uses its package structure for the interfaces. If the interface looks like org.freedesktop.MediaPlayer the Java class uses the package org.freedesktop. That means that in a folder found under /org/freedesktop/ a Java class named MediaPlayer.java has been created.

As every other needed Java class is created within the same folder, the script changes the dots of the interfacename to path separators, crops the last part, and compiles all Java Files found there (javac *.java).

Vlc player is ready to get controlled. The creation and the compilation of the Java classes need only be done once. You can ensure at the interface's path whether the files have been created and compiled.

Method invocation

Now it is possible to get the remote object and invoke actions on it. First the connection is established as described in Script 1, then the remote object is called via its bus name, its object path and the interface class that was just created. This example demonstrates three different kind of methods.

```
-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('org.freedesktop.MediaPlayer','Mp')
-- establish the connection to the session bus and get remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
mediaplayer = connection~getRemoteObject('org.mpris.vlc', '/Player', .Mp)

mediaplayer~pause                       -- without return value
say position = mediaplayer~PositionGet  -- returns value
mediaplayer~VolumeSet(50)               -- needs parameters

connection~disconnect
::requires BSF.CLS
```

Figure 10: Invoke actions of a mediaplayer program (vlc2.rxj)

All available methods can be looked up either with a dbus-debugger or in the created Java interface File.[31]

As you might have noticed, the program returns its track position in milliseconds. With Rexx it is very easy to convert this into a human readable string.

[31] The folder structure always equals its name, therefore the File can be found at following path /org/freedesktop/MediaPlayer.java.

```
position = mediaplayer~PositionGet
PARSE VALUE position/60000 WITH minutes '.' hsec
SAY 'position:' minutes 'minutes' LEFT(hsec*60 ,2) 'seconds'
```

Figure 11: Convert return value from mediaplayer

The method GetMetadata returns a DBusMap, the signature of the method in the MediaPlayer interface file shows, that it is just a Map[32] with Strings as keys and a Variant for the data. The following example demonstrates how values of such a Map can be extracted. Just add the following lines to the previous created script.

```
-- forwards a DBusMap to an internal routine
CALL saymap mediaplayer~GetMetadata

::routine saymap
USE ARG map
iterator = map~keySet()~iterator
DO WHILE (iterator~hasNext)
key = iterator~next
SAY key '=' map~get(key)~getValue
END
```

Figure 12: Handle a Dbus-Map (vlc3.rxj)

The mediaplayer does not expect extraordinary object types as parameters for method invocations and most methods do not even desire parameters. Thats the reason why this application is easy to interact with, it serves as an excellent starting point to collect experience with D-Bus.

Receiving signals

Not every application emits signals, for many applications it would be quite useless. The mediaplayer is not part of these, it emits signals about changes it its status (eg. paused, volume changed) and trackchanges.

To get informed about a signal it is necessary to create a Java class that implements the interface *'org.freedesktop.dbus.DBusSigHandler'*. Therefor a RexxProxy is very handsome. The class can be implemented right in the ooRexx script and the method of the DbusSigHandler can be overwritten with custom code.

[32] public Map<String,Variant> GetMetadata();

The signals are inner classes of the Java interface, during the compilation, they were complied as well. They can be identified by the '$' in their names, which separates the inner class from the outer. In this example the signal is named *'org.freedesktop.MediaPlayer$TrackChange'*. This script assumes that the necessary Java classes have already been created. (see figure 9 (vlc1.rxj))

```
-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('org.freedesktop.MediaPlayer','Mp')
-- establish the connection to the session bus
connection = .Connection~getConnection(box('int',.Connection~SESSION))
-- get the remote object
mediaplayer = connection~getRemoteObject('org.mpris.vlc', '/Player', .Mp)

.bsf~bsf.import('org.freedesktop.MediaPlayer$TrackChange', 'MPTC')
dir=.directory~new
signal=BsfCreateRexxProxy(.SignalHandler~new, dir, -
       'org.freedesktop.dbus.DBusSigHandler')
connection~addSigHandler(.MPTC, signal);

SAY 'Press enter to terminate'
PARSE PULL warte
connection~disconnect

-- SignalHandler implements DbusSigHandler and defines method handle
::class SignalHandler
::method handle
USE ARG signal
.bsf.dialog~messageBox('Signal:' signal~getName,'Signal arrived', information)
::requires BSF.CLS
```

Figure 13: Connection to a signal (vlc_signal.rxj)

The *directory* was created for the RexxProxy. With the help of this directory, information and references can be exchanged easily. In this nutshell example there is no need to provide any information through it. The script halts at the end and waits for user input, else it would terminate instantly and information about trackchanges would never arrive. The method *handle* draws up a simple messageBox displaying the name of the signal until its arrival.

If you do not want to wait for a track change to test the signal activity, it can be invoked by pressing the next or previous button on the player GUI.

3.5 Controlling a Text Editor

In this example I will demonstrate how to control the KDE advanced Text Editor (Kate)[33] which is part of a default KDE installation. This application is a little more tricky as we have to deal with process numbers and unusual object types.

If you try to control multiple instances of the vlc media player, you will only get reference to the first started instance. Kate on the other hand allows multiple instances running at the same time, each having a different process number. The script has to be adapted to handle with process numbers, as they are not fixed. When the program starts, it demands an available process number from the operating system which very likely differs every time.

With a dbus-debugger, the bus name *'org.kde.kate-1634'* can be identified. The number behind the bus name is its process number. In the object path section *'/Kate/Document/1'* can be found, the number denotes the documents currently opened in the text editor. The interface of this object is named *'org.kde.KTextEditor.Document'*. To obtain the process number the rexx script starts a program called *'pgrep'* on the command line[34] and processes its return value.

```
-- get the process number of a multiple instance application
cmd='pgrep -n -x -u "$USER" kate | rxqueue'
cmd
DO WHILE QUEUED()>0
  PARSE PULL pnumber
END
busname = 'org.kde.kate-'pnumber
objectpath = '/Kate/Document/1'
interfacename = 'org.kde.KTextEditor.Document'

-- create the interfaces with the given parameters
createinterface = 'CreateInterface -s -f' busname objectpath
createinterface

-- format the interfacename to a path
tmp = TRANSLATE(interfacename, '/' , '.')
path = SUBSTR(tmp, 1, tmp~LASTPOS('/'))

-- compile the created Java classes
compile = 'javac ./'path'*.java'
compile
```

Figure 14: Create and compile Java Files for a multi instance application (kate1.rxj)

[33] http://kate-editor.org/
[34] Information about pgrep can be obtained via the command 'man pgrep' on the command line.

As described in the previous example Java classes for the interface and the object types have been created and complied.

```
-- get the process number of a multiple instance application
cmd = 'pgrep -n -x -u "$USER" kate | rxqueue'
cmd
DO WHILE QUEUED()>0
  PARSE PULL pnumber
END
busname = 'org.kde.kate-'pnumber
objectpath = '/Kate/Document/1'
interfacename = 'org.kde.KTextEditor.Document'

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('org.kde.KTextEditor.Document','Te')
-- establish connection to the session bus and get remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
document = connection~getRemoteObject(busname,'/Kate/Document/1',.Te)

IF document~isEmpty THEN SAY 'Ok, we can go on' ELSE
DO
SAY 'Wait, there is already some text, exactly' document~totalCharacters 'characters'
END
connection~disconnect
::requires BSF.CLS
```

Figure 15: Invoke methods on a texteditor (kate2.rxj)

This example checks if the document is empty. As we access the object path */Document/1* we are controlling the document which was opened first in the text editor. In my case this is my ooRexx script, so I do not want to mess it up. With our knowledge we are ready to interact with multiple applications. A very handy program which also is part of every standard KDE distribution is called klipper. This application handles all strings in the clipboard, that means every string witch was marked with the mouse or copied via control-c.

Again we need the applications bus address and objectpath for the creation and the interfacename for the compilation. The script vlc1.rxj can be used as template, just change the parameters to:

```
busname = 'org.kde.klipper'
objectpath = '/klipper'
interfacename = 'org.kde.klipper.klipper'
```

Figure 16: Parameters for klipper program

After creation and compilation, the remote objects kate and klipper are ready to be used.

```
-- get the process number of a multiple instance application
cmd = 'pgrep -n -x -u "$USER" kate | rxqueue'
cmd
DO WHILE QUEUED()>0
  PARSE PULL pnumber
END
busname = 'org.kde.kate-'pnumber
objectpath = '/Kate/Document/1'
interfacename = 'org.kde.KTextEditor.Document'
busname2 = 'org.kde.klipper'
objectpath2 = '/klipper'
interfacename2 = 'org.kde.klipper.klipper'

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import(interfacename,'Te')
.bsf~bsf.import(interfacename2,'Kl')

-- establish connection to the session bus and get the remote objects
connection = .Connection~getConnection(box('int',.Connection~SESSION))
document = connection~getRemoteObject(busname,objectpath,.Te)
klipper = connection~getRemoteObject(busname2,objectpath2,.Kl)

IF document~isEmpty THEN SAY 'Ok, we can go on' ELSE
DO
CALL copytoclipboard document,klipper
END
connection~disconnect

::ROUTINE copytoclipboard
USE ARG document, klipper
SAY 'Wait, there is already some text, exactly' document~totalCharacters 'characters'
SAY 'Copying document text to the clipboard'
klipper~setClipboardContents(document~text)
document~clear
SAY 'Document cleared, Press Enter to retrieve content'
PARSE PULL wait
CALL retrievefromclipboard document,klipper

::ROUTINE retrievefromclipboard
USE ARG document, klipper
SAY 'retrieving the latest clipboard item and copy it back to the document'
document~clear
```

```
IF document~setText(klipper~getClipboardContents) THEN SAY 'retrieving done'

::requires BSF.CLS
```

Figure 17: Interaction between kate and klipper (kate3.rxj)

In this example, two remote objects are used, document and klipper. If the document is not empty, the routine 'copytoclipboard' is called which copies the content of the document to the clipboard and clears the content of the document. Now other functions can be tested on this empty document and the original content can be retrieved at the end.

'GetClipboardContents' always return the last item found in the clipboard, if it is not the correct document, the method *'getClipboardHistoryMenu'* returns an array of string elements, where the correct one can be identified and retrieved by its number (with *'getClipboardHistoryItem(number)'*).

The interface file of the editor [35]shows that the application interacts with 'Structs', there are four Structs extending the Struct class definition, numbered in order of their appearance. You might notice that all four of them implement exact the same code, the only difference is their name.[36] Nonetheless the correct Struct has to be identified.

```
-- get the process number of a multiple instance application
cmd = 'pgrep -n -x -u "$USER" kate | rxqueue'
cmd
DO WHILE QUEUED()>0
  PARSE PULL pnumber
END
busname = 'org.kde.kate-'pnumber
objectpath = '/Kate/Document/1'
interfacename = 'org.kde.KTextEditor.Document'

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import(interfacename,'Doc')

-- establish connection to the session bus and get remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
document = connection~getRemoteObject(busname, objectpath,.Doc)

.bsf~bsf.import('java.util.ArrayList','List')
.bsf~bsf.import('org.kde.KTextEditor.Struct3','Struct3')
list=.List~new
```

[35] File can be found here: /org/kde/KTextEditor/Document.java
[36] Normally the structs differ in their length or containing object types, therefore it is reasonable to call them with different names.

```
list~add("                ____ ")
list~add("  ___    ___  |  _ \  ___ __  ____  __")
list~add(" / _ \  / _ \ | |_) |/ _ \\ \/ /\ \/ /")
list~add("| (_) || (_) ||  _ <|  __/ >  <  >  < ")
list~add(" \___/  \___/ |_| \_\\___//_/\_\/_/\_\")

struct = document~endOfLine(1)
SAY 'line' struct~b 'ends at position' struct~a

document~clear
document~insertTextLines(.Struct3~new(10,10), list, 1)
connection~disconnect
```

Figure 18: Change content of texteditor document (kate4.rxj)

In this example, a Struct value is returned by the text editor and another Struct sent. As the Java file Struct1.java tells, there are two fields containing the values,[37] they can be accessed directly. The method *'insertTextLines'* needs a Struct for the position, a list for the lines to add, and a boolean value. The signature of Struct3 informs about the expected object type, in this case they are integer values. An Arraylist was chosen among available list containers and lines of an ASCII art[38] were added. With this knowledge, every method from the text editor can be called and return values can get processed.

The text editor does not emit signals, probably the programmer found no use for it. You could listen for text changes for example, by polling the characters in a document and compare them with an original value. A Signal would not make much sense, as it is obvious that text is entered in a text editor, therefore it would be useless to send signals, every time a character is typed or deleted.

3.6 Request a Bus Name

When a connection to the D-Bus is established, the bus daemon needs an address in order to route information from and to the connected application. This concept is quite similar to a host address in the Internet domain. These unique names are denoted as :1.98 for example. A program which wants to be retrieved easily by the user requests a well known bus name, for example 'org.kde.kate-1234'. But an application can request any bus name, if the name

[37] It does not make sense in this case to use a Struct, as the line number is already known, it would be sufficient to return the end of line in form of an integer value.
[38] ASCII Generator http://www.network-science.de/ascii/

is well formed[39] and not used by another application. As described in the previous chapters, an object path is necessary to identify the application correctly through their connection. If services should be provided from an application, it needs to define a Java class that implements 'org.freedesktop.DBus'. Within the created class, methods have to be defined.

```
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection', 'Connection')

connection = .Connection~getConnection(box('int', .Connection~SESSION))
app = .directory~new
testclass = BsfCreateRexxProxy(.OptionClass~new, app, 'org.freedesktop.DBus')

-- demands a name and exports an object
connection~exportObject('/Rexx', testclass)
connection~requestBusName('org.freedesktop.Rexx')

SAY 'Please watch out for the busname "org.freedesktop.Rexx"'
SAY 'Press enter to terminate'
PARSE PULL wait

connection~disconnect

-- simple class that only defines one method of 'org.freedesktop.DBus'
::CLASS OptionClass
::METHOD Hello
RETURN  'Hello! I am a rexx script, connected to the D-Bus via the Java binding'

::requires BSF.CLS
```

Figure 19: Export objects via D-Bus (exportObject.rxj)

In this example the name *'org.freedesktop.Rexx'* was chosen and the exported object was called *'/Rexx'*. A RexxProxy is used to implement the Java DBus class. Within the created ooRexx class, the method 'Hello' of the Java class is defined. If you use a D-Bus debugger, you will find the program listed on the session bus and the implemented method can be called and watched for its result.

3.7 Controlling Skype

Skype[40] is a program that enables users to effect telephone calls via Voice-over-IP. The Skype version used in this example is 2.2 Beta (2.2.0.35). The

[39] See chapter 1.2.6
[40] http://www.skype.com

program features (amongst other things) free calling to other Skype users[41], instant messaging[42] and file sending.[43]

The future of Skype on Linux is uncertain as Microsoft bought Skype and might not be interested in further development and/or support for this operating system. [Pro11]

Skype might not be connected to the D-Bus by default. The access can be activated in the preference menu under the sub-menu 'Public API'. There you will find a check-box which is called activate D-Bus. If you do not want to use the GUI you can start Skype on the command line with the switch: *--enable-dbus*

Skype does not offer service-access through an D-Bus interface like the programs mentioned above, but offers an *'Invoke'* method on its interface through which Skype's public API[44] gets accessed. Therefore it is not necessary to implement any object type (like a Struct described above), as only UTF-8 encoded Strings are exchanged to invoke methods and return results. [Sky11b]

Dependent of the Linux operating system it might be necessary to edit the skype configuration file.[45]

Look under */etc/dbus.1/system.d/* if there is a skype.conf. Create it or edit it to contain following content:

```
<!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
  <policy context="default">
    <allow own="com.Skype.API"/>
    <allow send_destination="com.Skype.API"/>
    <allow receive_sender="com.Skype.API"/>
    <allow send_path="/com/Skype"/>
  </policy>
</busconfig>
```

Figure 20: Skype configuration file (/etc/dbus-1/system.d/skype.conf)

Skype controls which applications are allowed to access its API, therefore the script needs to identify itself and Skype asks the user whether to allow or reject the connection.

[41] http://www.skype.com/intl/en/features/#
[42] http://www.skype.com/intl/en/features/allfeatures/instant-messaging/
[43] http://www.skype.com/intl/en/features/allfeatures/send-files/
[44] http://developer.skype.com/resources/public_api_ref.zip
[45] Worked out of the box in Ubuntu 11.04

This is effected with a Dialog (Figure 21) asking for confirmation. If Skype does not receive acknowledge in time, a timeout error arises and the connection is refused.

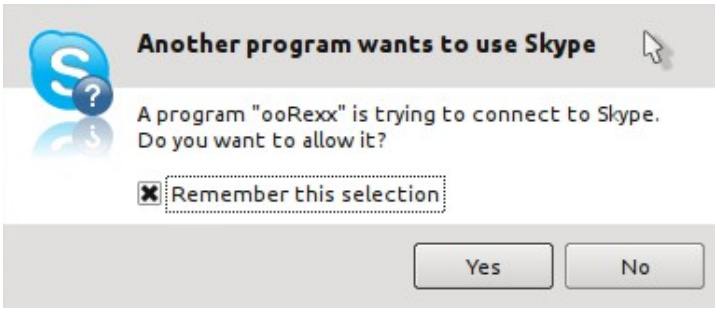

Figure 21: Skype access confirmation dialog

You can choose to remember the program for having the confirmation done automatically.[46]

Next the protocol for the API access has to be specified. Skype uses different protocols, they differ in their operating system and functionality -support. Protocol 7 is not fully supported in mentioned Linux Skype version. [Sky11b]

In this example protocol 5 will be used. It is therefore necessary to define it after the initial handshake was successful.

```
busname = 'com.Skype.API'
objectpath = '/com/Skype'
interfacename = 'com.Skype.API'

-- create the interfaces with the given parameters
createinterface = 'CreateInterface -s -f' busname objectpath
createinterface

-- format the interfacename to a path
tmp = TRANSLATE(interfacename, '/' , '.')
path = SUBSTR(tmp, 1, tmp~LASTPOS('/'))

-- compile the created Java classes
compile = 'javac ./'path'*.java'
compile

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('com.Skype.API','Sk')
-- establish connection to session bus and get remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
skype = connection~getRemoteObject(busname, objectpath ,.Sk)
```

[46] If you check remember this selection but receive errors during connection attempts, look up the submenu Pulbic API and check whether ooRexx is listed under allowed programs.

```
-- initiate a "handshake" and set up the skype api protocol to use
skype~Invoke('NAME ooRexx')
IF skype~Invoke('PROTOCOL 5') == 'PROTOCOL 5' THEN SAY 'Protocol 5 ready to be used'

connection~disconnect
::requires BSF.CLS
```

Figure 22: Accessing the Skype API (skype1.rxj)

The necessary interface is created and compiled, then the name ooRexx sent as identifier and protocol 5 defined for API access. Now a few functionalities of Skype can get tested.

Automated telephone calls can be very handy, imagine a use case where your scripts supervises system properties and in case of something bad happens, your system administrator gets called automatically. In the following example an automatic call to the test-user is effected. If the own userstatus is set to off-line, it needs to be activated before the call can be effected. As Skype needs a little time to get online, SysSleep is called.

```
busname = 'com.Skype.API'
objectpath = '/com/Skype'
interfacename = 'com.Skype.API'

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('com.Skype.API','Sk')

-- establish the connection to the session bus and get the remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
skype = connection~getRemoteObject(busname, objectpath ,.Sk)

skype~Invoke('NAME ooRexx')
skype~Invoke('PROTOCOL 5')

userstatus = skype~Invoke('GET USERSTATUS')
PARSE VALUE userstatus WITH 'USERSTATUS' status

if status \= 'ONLINE' THEN DO
SAY skype~Invoke('SET USERSTATUS ONLINE')
CALL SysSleep 1
END
SAY skype~Invoke('CALL echo123')

connection~disconnect
::requires BSF.CLS
```

Figure 23: Effect a test call with Skype (skype2.rxj)

If you want to make calls, check that you use the correct Skype username, not the name listed under fullname, or the name you might have chosen for a contact by yourself. (for example peter14352 instead of Peter.)

If you want to send messages you have to create a chat. The message which is returned by Skype,[47] contains the chat identification number. With this reference, messages can be send to the contact. In this example the testuser gets invited to a chat[48]. Change the contact to one of you own friends.

```
busname = 'com.Skype.API'
objectpath = '/com/Skype'
interfacename = 'com.Skype.API'
contact = 'echo123'

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('com.Skype.API','Sk')

-- establish connection to the session bus and get remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
skype = connection~getRemoteObject(busname, objectpath ,.Sk)

skype~Invoke('NAME ooRexx')
skype~Invoke('PROTOCOL 5')

response = skype~Invoke('CHAT CREATE' contact)
PARSE VALUE response WITH . chatid 'STATUS'

-- open the created chat an send a message
SAY skype~Invoke('OPEN CHAT' chatid)
SAY skype~Invoke('CHATMESSAGE' chatid 'Hello Skype!')

connection~disconnect
::requires BSF.CLS
```

Figure 24: Create a chat with Skype (skype3.rxj)

you can also create chats with multiple users, or add friends afterwards. It might also be quite useful to open a filetransfer window automatically to send log files to an administrator for example.

```
SAY skype~Invoke('CHAT CREATE' friend1 , friend2)
SAY skype~Invoke('ALTER CHAT' chatid 'ADDMEMBERS' friend)
SAY skype~Invoke('OPEN FILETRANSFER' friend)
```

Figure 25: Skype commands

[47] CHAT #ownname/$echo123;fe22fcbd114d377c STATUS DIALOG
[48] The testuser cannot receive messages, but you can see how to send messages.

The next example demonstrates how properties of a contact can be queried. Skype gets asked to list all friends, they get collected in an array and their birthday date gets compared with the actual date. If one (or more) of them has birthday, the message 'Happy Birthday!' is sent.

```
busname = 'com.Skype.API'
objectpath = '/com/Skype'
interfacename = 'com.Skype.API'

-- import needed Java classes
.bsf~bsf.import('org.freedesktop.dbus.DBusConnection','Connection')
.bsf~bsf.import('com.Skype.API','Sk')

-- establish connection to session bus and get remote object
connection = .Connection~getConnection(box('int',.Connection~SESSION))
skype = connection~getRemoteObject(busname, objectpath ,.Sk)

-- initial "handshake" and Protocol definition
skype~Invoke('NAME ooRexx')
skype~Invoke('PROTOCOL 5')

-- get names of added friends and collect them in an array
allfriends = skype~Invoke('SEARCH FRIENDS')
PARSE VALUE allfriends WITH 'USERS ' friends
array = friends~makeArray(', ')
.local~bdaywishes = 0

DO name OVER array
CALL checkbirthday skype, name
END

IF .local~bdaywishes == 0 THEN DO
SAY 'none of your friends has birthday today!'
END
ELSE SAY 'Happy Birthday has been sent to' .local~bdaywishes 'friends'

connection~disconnect

-- compare the actual date with the birthday date of a friend
::ROUTINE checkbirthday
USE ARG skype, name
datum=skype~Invoke('GET USER' name 'BIRTHDAY')
PARSE VALUE datum WITH . user . bday
day = RIGHT(bday,4)
today = RIGHT(DATE("S"),4)

IF day == today THEN DO
username = skype~Invoke('GET USER' name 'FULLNAME')
PARSE VALUE username WITH . 'FULLNAME' fullname
SAY fullname 'has birthday!'
CALL sendwishes skype, name
```

```
.local~bdaywishes += 1
END

-- send a message to a specified receiver
::ROUTINE sendwishes
USE ARG skype, name
response = skype~Invoke('CHAT CREATE' name)
PARSE VALUE response WITH . chatid 'STATUS'
SAY skype~Invoke('CHATMESSAGE' chatid 'Happy Birthday!')

::requires BSF.CLS
```

Figure 26: Send birthday wishes automatically (skype_birthday.rxj)

This example also demonstrates the excellent Rexx String treatment.

The list of friends is returned like: USERS friend1, friend2, friend3, friend4

It is easy for Rexx to make an array from such a String.

```
allfriends = skype~Invoke('SEARCH FRIENDS')
PARSE VALUE allfriends WITH 'USERS ' friends
array = friends~makeArray(', ')
```

Figure 27: Rexxs excellent String capabilities

3.8 Troubleshooting

If you get errors during executing of the sample scripts check following things.

Classpath

If you get an error which says: unable to load class 'org.freedesktop.dbus.DBusConnection', the java-dbus binding files could not be located.

- Look at the classpath via: echo $CLASSPATH if you cannot find the entry dbus.jar you have to export Java libraries to the classpath. Look at Figure 6 how to do that.
- The Java binding might not been installed correctly, try to reinstall it.

Java class error

if you get an error like [org.apache.bsf.BSFException: [EngineUtils.loadClass()] unable to load class ***]], the Java class is not available.

- Look if the execution directory contain a folder structure similar to the interface name. (interfacename=org.mpris.MediaPlayer, folder=/org/mpris)
- Look if this folder contains a Java file equal to the name of the interface (MediaPlayer.java) and check if this file has been compiled (there has to be a MediaPlayer.class file). If not, compile it via javac MediaPlayer.java.
- If there is no folder at all, ensure that you have executed the scripts for the interface creation of the desired application (for example vlc1.rxj)
- If no files are created look at spelling mistakes of parameters (use a D-Bus debugger for help).
- Check if you try to access a service that is not available anymore (for example accessing /Kate/Document/1 if the first document has already been closed).

Application error

The name *.*.* was not provided by any .service files

The desired application is not connected to the D-Bus.

- Check if the application is already started.
- If you know that an application has D-Bus support but is not available, look if the connection has to be enabled in its settings menu.
- Check if this application is a multi instance program, that means a reference to it needs a process number. (script kate1.rxj explains that)

If you get no reasonable answer from an application, look if two instance are running. You only get access to the first one, the second waits until the desired name has been given free again from the bus.

- close multiple instances

4 Roundup and Outlook

D-Bus provides many useful features for programmers. Applications can easily interact and exchange information. As D-Bus is supported by all Linux systems, it is the way to go to automate applications on this platform. There are bindings for many programming languages available, intending to map the D-Bus to the specific language. As there is no binding for ooRexx available up to now, the Java binding has to be used through BSF4ooRexx.

With the instructions provided in this paper, an ooRexx programmer can take use of the D-Bus capabilities. All features are accessed through the dbus-java binding. Once all necessary Java classes have been created, it is easy to use them via BSF4ooRexx, but the Java class creation and compilation itself is a burden to the java-dbus binding. As demonstrated in the appendix, the python binding do not need files to be generated. The object type handling suffers from the Java approach as well, as Java uses strictly typed object types. Therefore every object type exchanged through the D-Bus has to be defined, what might not be necessary for scripting languages like ooRexx. The python binding demonstrates its “object type guessing” capabilities in the appendix.

An ooRexx programmer should not abandon the advantages provided through a D-Bus connection. At this moment, it pays off learning how to use the java-binding through BSF4ooRexx. When there is a direct binding for ooRexx, the Java approach can gladly be passed back to Java programmers and ooRexx programmers can enjoy the easiness of use and the capabilities of their favored programming language on the D-Bus.

5 Bibliography

[Asc11] ASCII Generator

http://www.network-science.de/ascii/

[Bsf11] BSF4ooRexx

http://bsf4oorexx.sourceforge.net/

[Deb02] Debailleul: Guide Arexx AmigaOS 3.9

http://mat.debailleul.free.fr/amiga/DOC/**arexx**_times.pdf, 2002

[Dbb11] D-Bus Bindings

http://www.freedesktop.org/wiki/Software/DBusBindings

[Oor11] Open Object Rexx Reference
http://www.oorexx.org/docs/rexxref/book1.htm

[Fla10] Flatscher G. Rony: Reference Card Vienna Version of BSF4ooRexx 4.0.

http://sourceforge.net/projects/bsf4oorexx/files/beta/2011-06-18/BSF4ooRexx_install.zip/ refcardBSF4ooRexx.pdf

[Fre11a] Freedesktop Introduction

http://www.freedesktop.org/wiki/IntroductionToDBus

[Fre11b] Freedesktop Specification

http://dbus.freedesktop.org/doc/dbus-specification.html

[Fre11c] Freedesktop Dbus

http://www.freedesktop.org/wiki/Software/dbus

[Joh11a] Johnson Matthew: D-Bus programming in Java 1.5

http://dbus.freedesktop.org/doc/dbus-java/dbus-java.pdf

[Joh11b] Johnson Matthew: dbus-java API

http://dbus.freedesktop.org/doc/dbus-java/api/

[Kde07] KDE.org, DesktopCOmmunicationProtocol

http://api.kde.org/3.5-api/kdelibs-apidocs/dcop/html/index.html

[Mpr08] MPRIS D-Bus Interface Specification Version 1.0

http://www.mpris.org/1.0/spec

[Nok09] Nokia D-Bus Viewer

http://doc.qt.nokia.com/4.5/qdbusviewer.html

[Pal10] Palmieri John: D-Feet D-Bus debugger

http://live.gnome.org/DFeet/

[Pro11] Proffitt Brian: Microsoft's Skype acquisition may impact Linux users.

http://www.itworld.com/open-source/163509/microsofts-reported-skype-acquisition-may-impact-linux-users

[Sky11a] Skype Features

http://www.skype.com/intl/en/features/#

[Sky11b] Skype Public API

http://developer.skype.com/resources/public_api_ref.zip

[Tho11] Thompson Will: Bustle D-Bus profiler

http://willthompson.co.uk/bustle/

[Tro11] Trolltech: D-Bus introduction

http://doc.trolltech.com/4.2/intro-to-dbus.html

All links accessed on the 16.07.2011 for the last time.

6 Appendix

In this section a comparison of a method invocation, realized via the Python and the Java D-Bus Binding is drawn up. Please note that the process number of the remote object is hardcoded in this example. You have to identify the correct number if you want to test these scripts.

Python example:

```
import sys, dbus
kate = dbus.SessionBus().get_object('org.kde.kate-1979', '/Kate/Document/1')
kate.insertTextLines([10,10], ['line1','line2'],1,
dbus_interface='org.kde.KTextEditor.Document')
```

Figure 28: Appendix: Test.py

Java example:

```
import org.kde.KTextEditor.Struct3;
import org.freedesktop.dbus.*;
import java.util.ArrayList;
import org.freedesktop.dbus.exceptions.DBusException;
import org.kde.KTextEditor.Document;

public class Test {

    static DBusConnection conn = null;

    public static void main(String[] args) throws DBusException {
        conn = DBusConnection.getConnection(DBusConnection.SESSION);
        Document kate = conn.getRemoteObject("org.kde.kate-1979", "/Kate/Document/1",
              org.kde.KTextEditor.Document.class);
        ArrayList list = new ArrayList();
        list.add("line1");
        list.add("line2");
        kate.insertTextLines(new Struct3(10, 10), list, true);
    }
}
```

Figure 29: Appendix: Test.java

Java needs three files to do the same task, the above to invoke an action, an Interface for declaring the methods and an object type container called Struct.

```
package org.kde.KTextEditor;
import org.freedesktop.dbus.Position;
import org.freedesktop.dbus.Struct;
public final class Struct3 extends Struct
{
   @Position(0)
   public final int a;
   @Position(1)
   public final int b;
  public Struct3(int a, int b)
  {
   this.a = a;
   this.b = b;
  }
}
```

Figure 30: Appendix: Struct3.java

```
package org.kde.KTextEditor;

import java.util.List;
import org.freedesktop.dbus.DBusInterface;
import org.freedesktop.dbus.DBusInterfaceName;

@DBusInterfaceName("org.kde.KTextEditor.Document")

public interface Document extends DBusInterface
{
  public boolean clear();
  public boolean reload();
  public boolean save();
  public boolean saveAs();
  public boolean setTextLines(List<String> text);
  public boolean isEmpty();
  public int lineLength(int line);
  public Struct1 endOfLine(int line);
  public boolean insertText(Struct2 cursor, String text, boolean block);
  public boolean insertTextLines(Struct3 cursor, List<String> text, boolean block);
  public boolean cursorInText(Struct4 cursor);
  public boolean insertLine(int line, String text);
  public boolean insertLines(int line, List<String> text);
  public boolean removeLine(int line);
  public boolean setEncoding(String encoding);
  public void encoding();
  public boolean setText(String text);
  public String text();
  public int lines();
  public int totalCharacters();
}
```

Figure 31: Appendix: Document.java

It is needless to compare a script language like Python with Java in terms of generated code, it lies in the nature of scripting language to need less code, but this example illustrates how bindings to the D-Bus differ.

Signature: insertTextLines(Struct position, List text, Boolean block)

Python guesses the correct object types, therefore there is no need to define a list object, nor a Struct. As you can see, the Struct definition within the Java example even needs more lines of code than the whole python script.

Java needs an Interface to be declared[49], python solves this issues with reflection of the available interface. Therefore the Interface needs not be implemented, it is enough to reference it with its name as parameter.

Create a Java Swing GUI for method invocations of any application

The following example is an ooRexx script that creates all necessary interfaces and classes and sets up a Java swing GUI with buttons for each method to invoke and toggle buttons for signal connections.

The needed parameters are busname, interfacename, bus address and the object path of the desired application to connect. This script can be invoked by issuing following command on the command line (invoke without parameters informs about needed parameters).

rexx application.rxj org.mpris.vlc org.freedesktop.MediaPlayer 1 /Player

First all necessary Java classes are created, then they are compiled. Available methods form an interface are obtained via *~getDeclaredmethods*[50], all signals via *~getDeclaredClasses*. Actually the signals are not listed.[51] A Signal is defined as a inner class of an interface. But the signal connection was tested working in Java.

Not all return values from an application are displayed correctly as this part of the script is not finished yet. Method calls can only be effected within this pro-

[49] It would be sufficient for this example to declare only the 'insertTextlines' method, Nevertheless Python can access all methods if desired, therefore the Java equivalent was implemented feature complete.

[50] The methods equal, toString and hashcode are added as well, as they are unecessary at this moment, they can get excluded from the button creation. In this example they are not exluded.

[51] The execution of *getDeclaredClasses()* works in Java.

gram if only one parameter is necessary and if this parameter belongs to a standard object type. (String, Boolean, Integer)

Please take a look at the nutshell scripts in this paper in order to invoke method calls with more than one, non standard parameter. This demo script could possibly help, if you exploit some parts of it for your own application – at least this was the intention of adding it to this paper.

If you want to use this application for controlling purpose, it would be useful to change the beginning of the script and hardcode the necessary parameters, as on its first invocation, all Java classes are created and compiled. Additionally you would not need to pass parameters. If it is a multi instance application use a template like kate3.rxj to see how its process number can be obtained and exchange it with the commands in this script.

```
PARSE ARG object interfacename bus path
IF object~length<1 THEN
DO
SAY  Please add parameters
SAY 'first the BUSNAME. (eg. org.mpris.vlc)'
SAY 'followed by the INTERFACENAME. (eg. org.freedesktop.MediaPlayer)'
SAY 'then the BUS ADDRESS. (0 for systembus, 1 for sessionbus)'
SAY 'and then the OBJECT PATH. (eg. /Player)'
EXIT
END

-- creats the interfaces with the given parameters
cmd = 'CreateInterface -s -f' object path
--cmd

-- compiles the created classes
tmp = TRANSLATE(interfacename, '/' , '.')
compilepath = SUBSTR(tmp, 1, tmp~LASTPOS('/'))
compile = 'javac ./'compilepath'*.java'
--compile

SAY 'Interfaces created and compiled...'

.bsf~bsf.import('org.freedesktop.dbus.DBusConnection', "Connection")
.bsf~bsf.import(interfacename, "Interface")
connection = .Connection~getConnection(box('int',bus))
remoteobject = connection~getRemoteObject(object, path, .Interface)


methods = remoteobject~getclass~getDeclaredMethods
-- BSF4ooRexx does not return classes
signals = remoteobject~getclass~getDeclaredClasses
applicationname = object~substr(object~lastpos('.')+1)
app=.directory~new
app~application=remoteobject
app~signals=signals
```

```
app~connection=connection

system=bsf.loadClass('java.lang.System')
.local~newline=system~getProperty('line.separator')

--create all necessary handlers
rexxCloseEH=.RexxCloseAppEventHandler~new
rpCloseEH=BsfCreateRexxProxy(rexxCloseEH ,app , 'java.awt.event.WindowListener')
signalhandler=BsfCreateRexxProxy(.DBusSignalHandler~new, app, -
       'java.awt.event.ActionListener')
methodhandler=BsfCreateRexxProxy(.DBusMethodHandler~new, app, -
       'java.awt.event.ActionListener')
signal=BsfCreateRexxProxy(.SignalHandler~new, app, -
       'org.freedesktop.dbus.DBusSigHandler')
testclass=BsfCreateRexxProxy(.NewClass~new, app, 'org.freedesktop.DBus')

app~sighandler=signal
.bsf~bsf.import('javax.swing.JButton'                    , "Btn")
.bsf~bsf.import('javax.swing.JFrame'                     , "Frame")
.bsf~bsf.import('javax.swing.JLabel'                     , "Label")
.bsf~bsf.import('javax.swing.JPanel'                     , "Panel")
.bsf~bsf.import('javax.swing.JTabbedPane'                , "TabbedPane")
.bsf~bsf.import('javax.swing.JToggleButton'              , "TgBt")
.bsf~bsf.import('java.awt.FlowLayout'                    , "FlowLayout")
.bsf~bsf.import('java.awt.BorderLayout'                  , "BorderLayout")
.bsf~bsf.import('java.awt.Color'                         , "Color")
.bsf~bsf.import('java.awt.Dimension'                     , "Dimension")
.bsf~bsf.import('javax.swing.JTextArea'                  , "JTextArea")

-- exports an object and requests a name for the bus address
connection~exportObject('/Rexx',testclass)
connection~requestBusName('org.freedesktop.Rexx');

frame = .Frame~new('Control an application via DBus')
frame~setPreferredSize(.Dimension~new(510, 205))
flowlayout= .FlowLayout~new(.FlowLayout~LEFT)
methpanel = .Panel~new()~~setLayout(flowlayout)
signalpanel = .Panel~new()~~setLayout(flowlayout)

-- get all methods and signals from the application and add buttons
DO m OVER methods
PARSE VALUE m~toString WITH 'public final' rv method
PARSE VALUE method WITH methname '(' rest
name= methname~substr(methname~lastpos('.')+1)
cmd = name'('rest
methpanel~add(.Btn~new(name)~~addActionListener(methodhandler)~~setActionCommand(cmd))
END
i=1
DO s OVER signals
PARSE VALUE s~toString WITH ' ' inter '$' name
signalpanel~add(.TgBt~new(name)~~addActionListener(signalhandler)~~setActionCommand(i))
i+=1
END

-- finish setting up the Java GUI
tabbedpane = .TabbedPane~new()~~addTab('methods', methpanel)~~addTab('signals', -
       signalpanel)
appnamelabel = .Label~new(applicationname)
appnamepanel = .Panel~new()~~setBackground(.Color~WHITE)~~add(appnamelabel)
```

```
frame~getContentPane()~add(appnamepanel, .BorderLayout~NORTH)
frame~getContentPane()~add(tabbedpane, .BorderLayout~CENTER)
frame~~pack()~~setVisible(.true)~~toFront
frame~addWindowListener(rpCloseEH)
rexxCloseEH~waitForExit -- wait until we are allowed to end the program
CALL SysSleep .2

-- checks the returnvalue from the application and displays it
::routine answer
   USE ARG info
-- check for objecttype of return value
-- this method is not finished, object types have to be identified by their class
-- String comparison is not the best solution.

IF info==.nil THEN RETURN
 PARSE VALUE info WITH name '@' number

IF pos('.DBusMap',name)>0 THEN CALL handlemap info
IF pos('.Variant',name)>0 THEN SAY "Received a Variant, sorry not finished yet."
IF pos('.Tuple', name)>0 THEN SAY "Received a Tupel, sorry not finished yet."
IF pos('.Struct',name)>0 THEN CALL handlestruct info
IF number<1 THEN .bsf.dialog~messageBox('Answer:' name,'Answer form application', -
       information)

::routine handlemap
USE ARG info
iterator = info~keySet()~iterator
text = .JTextArea~new
DO WHILE (iterator~hasNext)
key = iterator~next
text~append(key '=' info~get(key)~getValue .newline)
END
.bsf.dialog~messageBox(text,'Answer from application', information)

::routine handlestruct
USE ARG info
temp = info~getParameters
i=1
text = .JTextArea~new
DO WHILE i<temp~size
text~append(temp[i] .newline)
i+=1
END
.bsf.dialog~messageBox(text,'Answer from application', information)


::class RexxCloseAppEventHandler
::method init        /* constructor */
  EXPOSE closeApp
  closeApp  = .false          -- if set to .true, then it is safe to close the app
::attribute closeApp          -- indicates whether app should be closed
::attribute application get
::method unknown              -- intercept unhandled events, do nothing
::method windowClosing        -- event method (from WindowListener)
  EXPOSE closeApp
  closeApp=.true              -- indicate that the app should close
::method waitForExit          -- method blocks until attribute is set to .true
  EXPOSE closeApp
```

```
  guard on when closeApp=.true

::class DBusSignalHandler
::method actionPerformed
  USE ARG eventObject, args
  selected = eventObject~getSource~isSelected
  index = eventObject~getActionCommand
  connection = args~userdata~connection
  signals = args~userdata~signals
  sighandler = args~userdata~sighandler

  IF selected THEN DO
   connection~addSigHandler(signals[index], sighandler)
   SAY 'Signalhandler added'
   END
  ELSE DO
   connection~removeSigHandler(signals[index], sighandler)
   SAY 'Signalhandler removed'
  END

::class DBusMethodHandler

::method actionPerformed
  USE ARG eventObject, args
  application = args~userdata~application
  command = eventObject~getActionCommand

  PARSE VALUE command WITH cmd '(' params
  IF (params>0) THEN DO
  val = .bsf.dialog~inputBox('Signature: ('params,'','')
  IF val==.nil THEN RETURN
  IF val~length<1 THEN RETURN
  CALL answer application~send(cmd,val)     -- calls the desired action and
  END                                       -- forwards return values to answer
  ELSE CALL answer application~send(cmd)

::class SignalHandler
::method handle
  USE ARG signal
  .bsf.dialog~messageBox('Signal:' signal~getName,'Signal arrived', information)

::class NewClass
::method Hello
  RETURN 'Hello! I am accessing the D-Bus via the java-dbus binding and BSF4ooRexx'
::requires BSF.CLS
```

Figure 32: Sample script D-Bus control gui (app.rxj)