Vienna University of Economics and Business

Institute for Management Information Systems

Course 0208, Projektseminar aus Wirtschaftsinformatik (Schiseminar)

D-Bus Language Binding for ooRexx: An Introduction on Nutshell Examples

Richard Lagler, 1152821

Supervisor: ao. Univ.Prof. Mag. Dr. Rony G. Flatscher

February 20, 2015

Declaration of Authorship

"I do solemnly declare that I have written the presented research thesis by myself without undue help from a second person others and without using such tools other than that specified.

Where I have used thoughts from external sources, directly or indirectly, published or unpublished, this is always clearly attributed.

Furthermore, I certify that this research thesis or any part of it has not been previously submitted for a degree or any other qualification at the Vienna University of Economics and Business or any other institution in Austria or abroad."

Date:

Signature:

Contents

List of Figures

List of Tables

Listings

Abstract

"D-Bus is a message bus system, a simple way for applications to talk to one another. In addition to interprocess communication, D-Bus helps coordinating process lifecycle; it makes it simple and reliable to code a "single instance" application or daemon, and to launch applications and daemons on demand when their services are needed."[see FreeD14a] In order to do so the paper first introduces the D-Bus concepts and demonstrates the language binding for ooRexx. Based on nutshell examples the current approaches are demonstrated and should enable ooRexx programmers to use it for their work.

Key words: D-Bus, Open Object Rexx (ooRexx), Freedesktop, language binding, Linux, bus, messaging

1 Introduction

"D-Bus is being used on Linux systems for kernel programs to communicate with other parts of the Linux system and to emit signals for further controls. These particular communications are taking place over a D-Bus "system" message daemon ("system bus") that gets started when Linux boots." [see Flat11, I] In addition, whenever a user logs into a Linux system and thereby starts a new session a separated bus "session" message daemon ("session bus") gets started. This D-Bus allows any application in the user's session to interact with each other which means some remote controlling action. [see Flat11, I]

Via a D-Bus connection, every application can provide services which can get accessed from another application. Transported object types get coded from their native implementation to the bus specification, routed through the bus to a specific receiver. As a consequence the user can hand over commands containing different objects using ooRexx which may enhance the remote controlled applications' action.[see FreeD14b]

This article is structured such that it gives first an introduction to D-Bus methodology scripted with ooRexx and continues with investigations on common known applications e.g. VideoLAN Player, Kopete and Bluetooth. Nutshell examples are structured based on its complexity whether signals are only sent or additionally interpreted. Complementary, the last example concludes with an ooRexx D-Bus server which provides methods via the session bus and listens continuously.

1.1 History

"D-Bus was first built to replace the CORBA-like component model underlying the GNOME desktop environment. Similar to DCOP (which is used by KDE), D-Bus is set to become a standard component of the major free desktop environments for GNU/Linux and other platforms. A GNOME environment normally runs two kinds of buses: a single system bus for miscellaneous system-wide communication, e.g. notifications when a new piece of hardware is hooked up; and a session bus used by a single user's ongoing GNOME session. A session bus normally carries traffic under only a single user identity, but D-Bus is aware of user identities and does support flexible authentication mechanisms and access controls. The system bus may see traffic from and to any number of user identities". [see FreeD14b]

1.2 Open Object Rexx

"The scripting language "Open Object Rexx (ooRexx)" is an object-oriented extension of the REXX scripting language, which was intentionally designed as a "human-oriented" language. A brief overview of its history and features is given. The language can be briefly characterized as:

- an interpreted language,
- having an easy, pseudo-code like syntax,
- being dynamically typed (REXX: "everything is a string", ooRexx: "everything is an object"),
- caseless (everything outside of quotes gets uppercased by the interpreter before execution), possessing an explicit message operator, the tilde (~); left of the tilde is the receiving object, right of it the method name and, optionally (in round brackets) the arguments supplied with the message,
- drawing concepts from Smalltalk,
- possessing an easy to use, yet powerful C++ API.

ooRexx was originally developed by IBM which handed over its source-code to the non-profit special interest group "Rexx Language Association (RexxLA)" for opensourcing and further developing the language." [see Flat11, 8]

1.3 Language Bindings

"Application Programming Interfaces for D-Bus, or bindings, are available in several languages; typically one per language, but not necessarily. Each presents its own API as suits the language, hiding the details of working with D-Bus from the programmer to different extents. The ideal is to fit the D-Bus API into the native language and libraries as naturally as possible." [see FreeD14b] Using D-Bus should feel more like object-oriented programming than like communication. In some bindings, a programmer may hardly notice that D-Bus is there at all.

When it comes to explanations on nutshell examples it will be proved how straightforward D-Bus objects can be used with ooRexx. The binding works as a native conjunction to D-Bus what creates the feeling to deal with ooRexx objects only.

"When that happens, a program that uses D-Bus to communicate will for the most part look as if the counterparts it communicates with were regular components (libraries, modules, packages, objects, functions; whatever the language uses) of the program itself. This is also why some aspects of D-Bus that may seem very basic can differ greatly depending on programming language." [see FreeD14b]

2 Overview of D-Bus

In this chapter an introduction on the infrastructure of D-Bus is given.

2.1 Buses

"There are two major components to D-Bus: a point-to-point communication "dbus" library, which in theory could be used by any two processes in order to exchange messages among themselves; and a "dbus" daemon. The daemon runs an actual bus, a kind of "street" that messages are transported over, and to which any number of processes may be connected at any given time. Those processes connect to the daemon using the library, and it probably wouldn't make much sense to use the library for anything else." [see FreeD14b]

Applications can either be called by their unique name representation (for example: 35-731) or with a name (e.g. org.kde.KTextEditor). A program can request any name to be easily identifiable if the name is not already registered. [see Marg11, 10]

2.2 Addresses

"Every bus has an address describing how to connect to it. A bus address will typically be the filename of a Unix-domain socket such as "/tmp/.hiddensocket," but it may also be a TCP port where a bus daemon is listening on an IP-domain socket." [see FreeD14b]

All methods that are made available through the D-Bus have to be specified in interfaces which means a developer has to define interfaces' addresses. Since programmers can choose the applications' names he or she can also register an appropriate path/address to the application based on its complexity and structure.

"From a functional standpoint, the primary purpose of object paths is simply to be a unique identifier for an object. The "hierarchy" implied the path structure is almost purely conventional. Applications with a naturally hierarchical structure will likely take advantage of this feature while others may choose to ignore it completely." [see FreeD14b]

2.3 Signature Strings

"D-Bus uses a string-based type encoding mechanism called signatures to describe the number and types of arguments requried by methods and signals. Signatures are used for interface declaration/documentation, data marshalling, and validity checking. Their string encoding uses a simple, though expressive, format and a basic understanding of it is required for effective D-Bus use. The table below lists the fundamental types and their encoding characters." [see FreeD14b]

Data Type	**Type Indicator**	**Comment**
array	a	If sequence of 'a's, then each 'a' stands for one dimension, followed by the element type indicator of the array.
boolean	b	'0' (false) or '1' (true)
byte	y	8-bit unsigned integer
double	d	IEEE 754 double
int16	n	16-bit signed integer
int32	i	32-bit signed integer
int64	x	64-bit signed integer
objpath	o	Must start with a slash (/).
signature	g	May consist of type indicators only.
string	s	Must be encoded as modified UTF-8.0
uint16	q	16-bit unsigned integer
uint32	u	32-bit unsigned integer
uint64	t	64-bit unsigned integer
unix_fd	h	Available only, if using Unix socket transportation.
variant	v	A container type which includes the signature of the encoded value.
structure	(..)	A container type. Parentheses may contain any types.
map/dict	a{s..}	A container type. Map/dictionary, index is always a string, value can be of any type.

Table 1: Signature Strings
[see Flat11, 4]

2.4 Exchange of Messages

As introduced a D-Bus message daemon allows any processes to send each other messages. The broker receives and forwards messages where D-Bus provides the communication infrastructure (The broker is the D-Bus message daemon). A message could be addressed via its well-known bus name `"org.freedesktop.DBus"`; in order to exchange messages, a process needs to tell the broker the destination address. [see Flat11, 5] See also chapter 2.2 for details on addresses.

2.5 Interfaces

"D-Bus interfaces define the methods and signals supported by D-Bus objects. In order to make use of a D-Bus interface it must be known to remote users. This interface definition may be hard coded into an application or may be queried at run time through the D-Bus introspection mechanism." [see FreeD14b]

For instance, the standard `Introspection` interface is `"org.freedesktop.DBus.Introspectable"`.

Methods may accept any number of arguments and may return any number of return values, including none. Methods and properties can be designed with the help of signature strings but are strictly structured as described. [see FreeD14b]

2.5.1 Introspect Member

With the help of the `Introspection` interface an XML encoded string can be submitted in order to publish interface names and all their members which can be of type method, signal or property.

Returns	**Message Type**	**Member Name**
s	method	Introspect()

Table 2: Interface: "org.freedesktop.DBus.Introspectable"
[see Flat11, 6]

Types of the arguments of methods, signals or properties are encoded with indicators given in table 1 where methods that do not return any value will document this with an empty string as their signature. Signals, being one-way (broadcast) messages may carry arguments, but will never return a value. [see Flat11, 6]

2.5.2 Get & Set Members

To access properties, getter and setter methods are provided in interface `"org.freedesktop.DBus.Properties"` as shown in table 3.

Returns	Message Type	Member Name
v	method	Get(ss)
	method	Set(ssv)
a{sv}	method	GetAll(s)

Table 3: Interface: "org.freedesktop.DBus.Properties"
[see Flat11, 7]

2.5.3 D-Bus Message Daemon

As introduced the message broker owns the bus named `"org.freedesktop.DBus"`. Using the Introspect member, a returned string describes all published interfaces and their members listed in table 4

Returnse	Message Type	Member Name
	method	AddMatch(s)
ay	method	GetAdtAuditSessionData(s)
ay	method	GetConnectionSELinuxSecurityContext(s)
u	method	GetConnectionUnixProcessID(s)
u	method	GetConnectionUnixUser(s)
s	method	GetId( )
s	method	GetNameOwner(s)
s	method	Hello( )
as	method	ListActivatableNames( )
as	method	ListNames( )
as	method	ListQueuedOwners(s)
b	method	NameHasOwner(s)
u	method	ReleaseName(s)
	method	ReloadConfig( )
	method	RemoveMatch(s)
u	method	RequestName(su)
u	method	StartServiceByName(su)
	method	UpdateActivationEnvironment(ass)
	signal	NameAcquired(s)
	signal	NameLost(s)
	signal	NameOwnerChanged(sss)

Table 4: D-Bus Members
[see Flat11, 7]

2.5.4 Private D-Bus Server

"The D-Bus interprocess communication infrastructure allows programmers to use the infrastructure without the help of a D-Bus message daemon. Such servers are called private D-Bus servers. This allows for simple client/server applications where any process having a connection to the private D-Bus server can communicate with it using D-Bus messages." [see Flat11, 8]

3 Language Bindings

"In computing, a binding from a programming language to a library or operating system service is an application programming interface (API) providing glue code to use that library or service in a particular programming language." [see Wiki14] There are many bindings available for D-Bus; the most common known are: Python, Java, Qt4, Perl, C++, PHP, .NET, Ruby and ooRexx. Since the paper focuses on the ooRexx language binding only a short introduction on other solutions is given.

3.1 ooRexx

This chapter introduces the setup of the language binding for ooRexx.

3.1.1 Setup

In this article all nutshell examples are developed in a Linux environment, Kubuntu 14.10.

If you do not already use a Linux operating system or 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. [see Marg11, 19]

As D-Bus and ooRexx are also available for Windows, you can give it a try for testing purposes only.

Firstly, Open Object Rexx 4.2.0 or higher has to be installed downloadable at
http://sourceforge.net/projects/oorexx/files/oorexx/4.2.0/

Secondly, DBus4ooRexx has to be installed. The language binding for ooRexx can be downloaded at:

- **DBus4ooRexx Download:**
  https://sourceforge.net/projects/bsf4oorexx/files/GA/sandbox/dbusoorexx/
- **DBus4ooRexx Source Code:**
  https://sourceforge.net/p/bsf4oorexx/code/HEAD/tree/sandbox/rgf/misc/dbusoorexx/
- **32- and 64-Bit DBus for Windows (for testing purposes only)**
  http://wi.wu.ac.at/rgf/rexx/orx22/work/

In order to run the nutshell examples successfully the following applications have to be installed manually:

- Bluetooth driver and connection manager
- Kopete
- VideoLAN Player

3.1.2 ooRexx Class "DBus": Getting a Connection

In order to establish a connection the ooRexx class `DBus` has to be used named `"dbus.cls"`.

The class methods `connect` and `new` allow creating a new D-Bus connection by giving an address to connect to. Additionally the class methods `system` and `session` can be used as well as shown:

- `new(server_address | "system" | "session" )`
- `connect(server_address | "system" | "session" )`
- `system`
- `session`

To send messages from ooRexx the method `message` must be used where its first argument can be either `call` or `signal`. [see Flat11, 10]

"The method `listener` allows to add or remove an ooRexx listener object, as well as getting a list of currently registered ooRexx listener objects. An ooRexx listener object gets wrapped up and stored in a `"DBusListener"` object.

The method `serviceObject` allows to add or remove an ooRexx service object, as well as getting a list of currently registered ooRexx service objects.

The method `busName` allows to request and to release the ownership of a bus name." [see Flat11, 10]

3.1.3 Hello from D-Bus with ooRexx

Traditionally, the first example will be a Hello World script where the method `message` in listing 1 on bus `"org.freedesktop.Notifications"` is used. The notification in figure 1 is displayed.

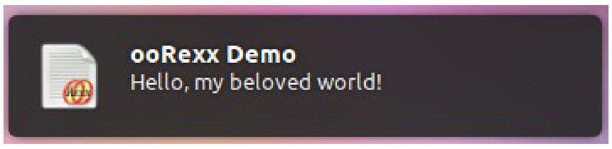

ooRexx Demo
Hello, my beloved world!

Figure 1: A Hello World from ooRexx with Method Message

```
conn=.dbus~session /* get connection to the session message bus */

/* define message arguments */
busName ="org.freedesktop.Notifications"
objectName ="/org/freedesktop/Notifications"
interfaceName ="org.freedesktop.Notifications"
memberName ="Notify"
replySignature="u" /* uint32 */
argSignature ="susssasa{sv}i"
/* string, uint32, string, string, string, array of string, dict of variants, int3 */

id=conn~message("call",busName,objectName,interfaceName,memberName, -
        replySignature,argSignature,"An ooRexx App", , -
                "oorexx", "ooRexx Demo", "Hello, my beloved world!", , , -1)

::requires "dbus.cls" /* get DBus support */
```

Listing 1: helloworld.rex

3.1.4 Remote Service Objects

As introduced in chapter 1 the language binding follows the human orientation of ooRexx. With the help of ooRexx D-Bus proxy the supplementation of correct signatures will work automatically by using remote objects.

Additionally, properties can also be used as native ooRexx attributes rather than investigating the `"org.freedesktop.Properties"` interface all the time.

The class `DBusProxyObject` automatically introspects remote objects and uses its interface definition. When developers send messages to the remote D-Bus service object, no type information needs to be supplied as this is already known by the proxy object.

Remote objects can be accessed with the method `getObject(busName, ObjectPath)`. [see Flat11, 12]

In listing 2 the Hello World notification is displayed with the help of the ooRexx remote service object and method notify. [see Flat11, 13]

```
/* get access to remote object */
o=.dbus~session~getObject("org.freedesktop.Notifications","/org/freedesktop/Notifications")
id=o~notify("An ooRexx App", , "oorexx", "ooRexx Demo", "Hello, my beloved world!", , , -1)

::requires "dbus.cls" /* get DBus support */
```

Listing 2: helloworldrso.rex

3.1.5 Listener for D-Bus Signals

The class DBusListenerObject is a wrapper class which stores the ooRexx object to which signal messages should get forwarded to (attribute listenerObject). Additionally, it allows storing an interface name (attribute interface) and/or a signal name (attribute signalName), which can be used for additional filters.

The method listener in class DBus wraps an ooRexx listener object in an instance of the DBusListenerObject class. Listing 3 depicts an ooRexx programm, which connects to the session message daemon and adds an ooRexx listener object (method listener). The D-Bus broker will forward all signal messages (method match with argument type="signal") which enables a discovering service for all messages broadcasted via the session bus daemon on the command line. [see Flat11, 14]

```
signal on halt /* intercept ctl-c (jump to label 'halt:' below) */

conn=.dbus~session /* get the "session" connection */
conn~listener("add", .rexxSignalListener~new) /* add the Rexx listener object */
conn~match("add", "type='signal'", .true) /* ask for any signal message */

say "Hit enter to stop listener..."
parse pull answer /* wait for pressing enter */
halt: /* a label for a halt condition (ctl-c) */
say "closing connection."

conn~close /* close connection, stops message loop */

::requires "dbus.cls" /* get dbus support for ooRexx */

::class RexxSignalListener /* just dump all signals/events we receive */
::method unknown /* this method intercepts all unknown messages */
  use arg methName, methArgs
  slotDir=methArgs[methArgs~size] /* last argument is slotDir */
  say "-->" pp(slotDir~messageTypeName) pp(slotDir~interface) -
        pp(slotDir~member)", nrArgs="methArgs~items-1



  do i=1 to methArgs~items-1
    say " argument #" i":" pp(methArgs[i])
  end

  say "-"~copies(79)

::method NameOwnerChanged /* demo how to directly intercept a signal */
  use arg name, old, new, slotDir
  say "==> NameOwnerChanged:" "Name:" pp(name)", OldOwner:" pp(old) )) -
        ", NewOwner:" pp(new)
  say "-"~copies(79)

::routine pp /* "pretty print": enclose string value with square brackets */
  parse arg value
  return "["value"]"
```

Listing 3: signalListener.rex

In Chapter 5 some nutshells examples are provided to view the output of the ooRexx listener and how it can be used to react on specific signals. The listener enables actions based on conditions which can be somehow useful in production environments.

3.2 Python

The Python binding can be installed using the dbus-python package downloadable at
D-Bus-Python Download:
http://dbus.freedesktop.org/releases/dbus-python/

3.3 Java

The Java language binding is named dbus-java and is currently available at version 2.7.
D-Bus-Java Download:
http://dbus.freedesktop.org/releases/dbus-java/dbus-java-2.7.tar.gz

There is also a well-structured documentation available at:
http://dbus.freedesktop.org/doc/dbus-java/dbus-java/.

4 Exploring D-Bus-Services

Communication through the buses can be monitored with the help of the tool qdbus-monitor issued in a command line. As a result, messages are listed with information about sender and receiver (if any). Due to the amount of information while using the system it might be quite stressful to analyse information coming from the tool. However, it is possible.

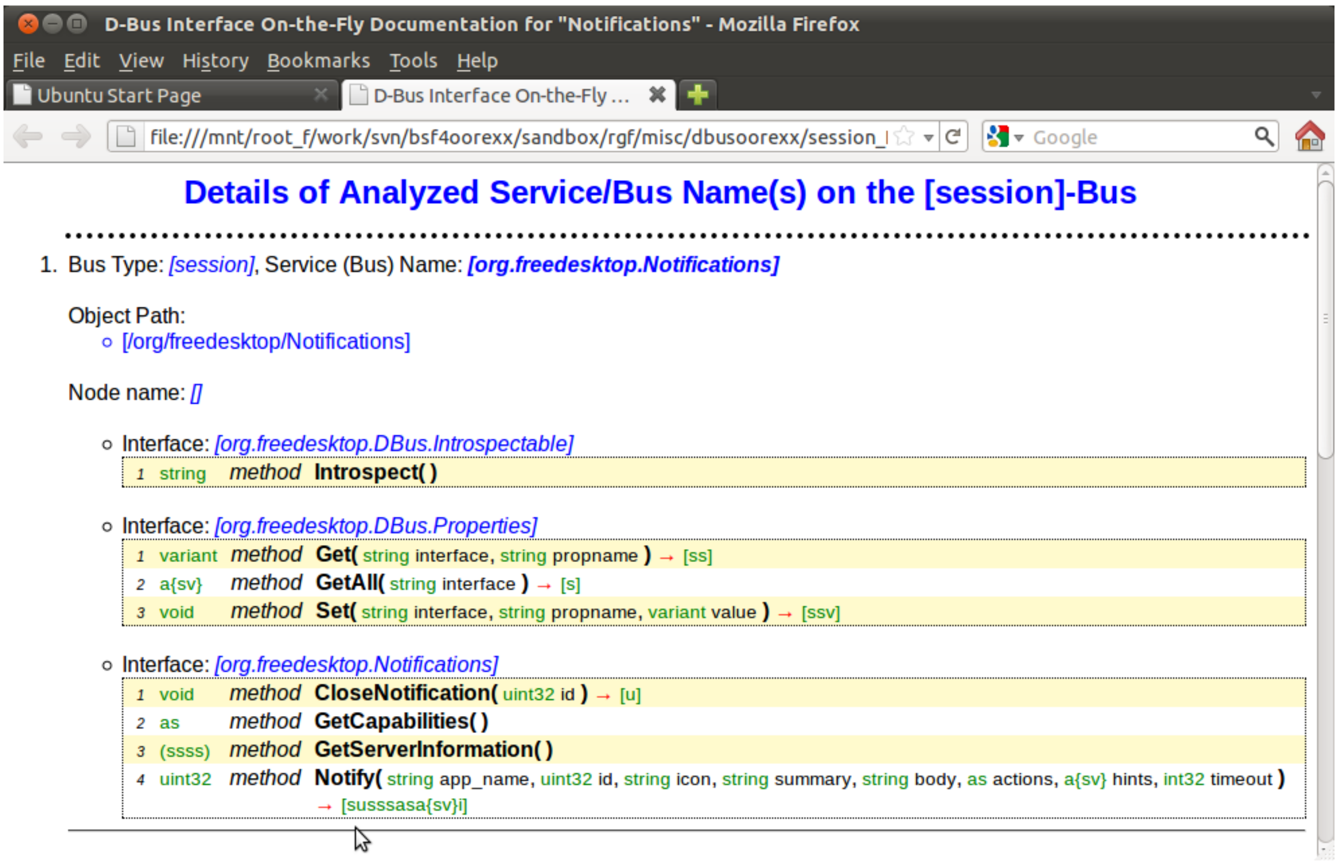
There are different approaches available to consult D-Bus interfaces, also with a graphical user interface, for instance D-Feet and qdbusviewer. [see Marg11, 13 f.]

Since it seems to be quite hard to research documentation manually the ooRexx language binding comes along with the tool "dbusdoc.rex" which generates a nice to view documentation with HTML.

4.1 D-Bus On-the-Fly-Documentation "dbusdoc.rex"

Although it is possible to research some D-Bus interfaces' documentation with the help of tools e.g. D-Feet and qdbusviewer live, it will be an incredible help to use the utility "dbusdoc.rexx" which introspects services and its interfaces using object path discovery. The tool determines interface definitions and groups these objects path into alphabetical order which will be saved into a HTML file. In order to run the built-in tool of the ooRexx D-Bus language binding the syntax is: `dbusdoc.rex [session|system] [services.bus.name]`

If no arguments are given, the command line tool lists all available service names. Figure 2 displays the documentation for the service `"org.freedesktop.Notifications"`. Please be aware that the documentation may be different depending on the Linux distribution in use. [see Flat11, 21 f.]


D-Bus Interface On-the-Fly Documentation for "Notifications" - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Ubuntu Start Page
D-Bus Interface On-the-Fly ...
file:///mnt/root_f/work/svn/bsf4oorexx/sandbox/rgf/misc/dbusoorexx/session_
Google

Details of Analyzed Service/Bus Name(s) on the [session]-Bus

1. Bus Type: *[session]*, Service (Bus) Name: ***[org.freedesktop.Notifications]***

   Object Path:
   - [/org/freedesktop/Notifications]

   Node name: *[]*

   - Interface: *[org.freedesktop.DBus.Introspectable]*

1	string	*method*	**Introspect( )**

   - Interface: *[org.freedesktop.DBus.Properties]*

1	variant	*method*	**Get(** string interface, string propname **)** → [ss]
2	a{sv}	*method*	**GetAll(** string interface **)** → [s]
3	void	*method*	**Set(** string interface, string propname, variant value **)** → [ssv]

   - Interface: *[org.freedesktop.Notifications]*

1	void	*method*	**CloseNotification(** uint32 id **)** → [u]
2	as	*method*	**GetCapabilities( )**
3	(ssss)	*method*	**GetServerInformation( )**
4	uint32	*method*	**Notify(** string app_name, uint32 id, string icon, string summary, string body, as actions, a{sv} hints, int32 timeout **)** → [susssasa{sv}i]

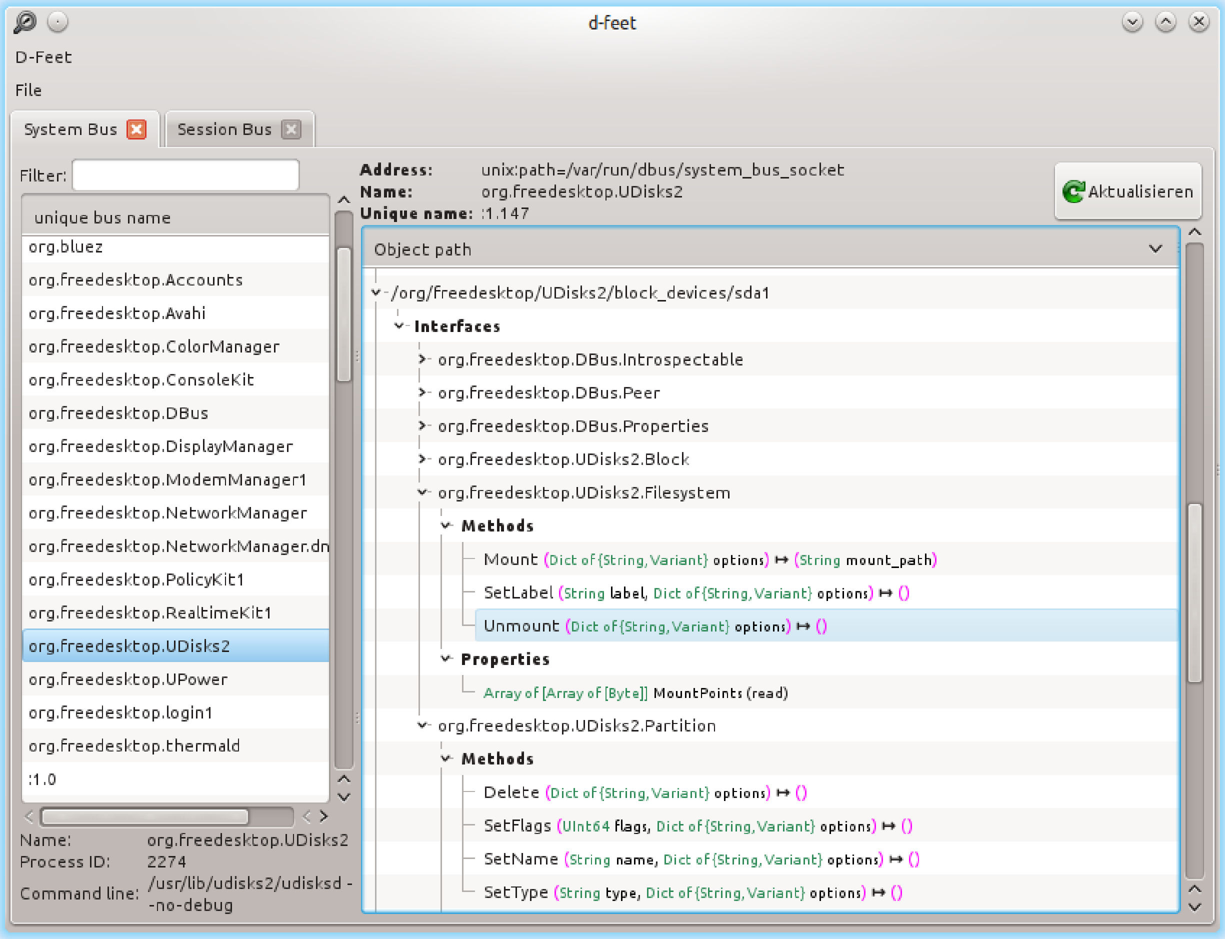
Figure 2: On-the-Fly-Documentation "dbusdoc.rex"

4.2 D-Feet

The tool D-Feet provides a user interface and enables users to search for services and drill down into interfaces' definition. To use the tool basic know-how about object paths is necessary in order to be able to consult the interfaces. It might be easy to use if the user knows the service name to introspect as the tool provides a live introspection. However, the tool does not represent a full documentation easy to read since it is a viewer accessing the system in real-time.

In figure 3 on the left side of the panel, all discovered applications are listed divided into system and session bus. Once an application is selected all declared paths and interfaces are listed which can be drilled-down easily. [see Marg11, 13 f.]

In order to prove the system-wide existence of D-Bus services figure 3 shows the introspection of Linux disk-management's interface "`org.freedesktop.UDisks2`". There are for instance the methods `Mount` and `SetLabel` available.



```
unique bus name
org.bluez
org.freedesktop.Accounts
org.freedesktop.Avahi
org.freedesktop.ColorManager
org.freedesktop.ConsoleKit
org.freedesktop.DBus
org.freedesktop.DisplayManager
org.freedesktop.ModemManager1
org.freedesktop.NetworkManager
org.freedesktop.NetworkManager.dn
org.freedesktop.PolicyKit1
org.freedesktop.RealtimeKit1
org.freedesktop.UDisks2
org.freedesktop.UPower
org.freedesktop.login1
org.freedesktop.thermald
:1.0

Name:          org.freedesktop.UDisks2
Process ID:    2274
Command line:  /usr/lib/udisks2/udisksd -
               -no-debug
```

```
Address:      unix:path=/var/run/dbus/system_bus_socket
Name:         org.freedesktop.UDisks2
Unique name:  :1.147

Object path
/org/freedesktop/UDisks2/block_devices/sda1
  Interfaces
    org.freedesktop.DBus.Introspectable
    org.freedesktop.DBus.Peer
    org.freedesktop.DBus.Properties
    org.freedesktop.UDisks2.Block
    org.freedesktop.UDisks2.Filesystem
      Methods
        Mount (Dict of {String, Variant} options) ↦ (String mount_path)
        SetLabel (String label, Dict of {String, Variant} options) ↦ ()
        Unmount (Dict of {String, Variant} options) ↦ ()
      Properties
        Array of [Array of [Byte]] MountPoints (read)
    org.freedesktop.UDisks2.Partition
      Methods
        Delete (Dict of {String, Variant} options) ↦ ()
        SetFlags (UInt64 flags, Dict of {String, Variant} options) ↦ ()
        SetName (String name, Dict of {String, Variant} options) ↦ ()
        SetType (String type, Dict of {String, Variant} options) ↦ ()
```

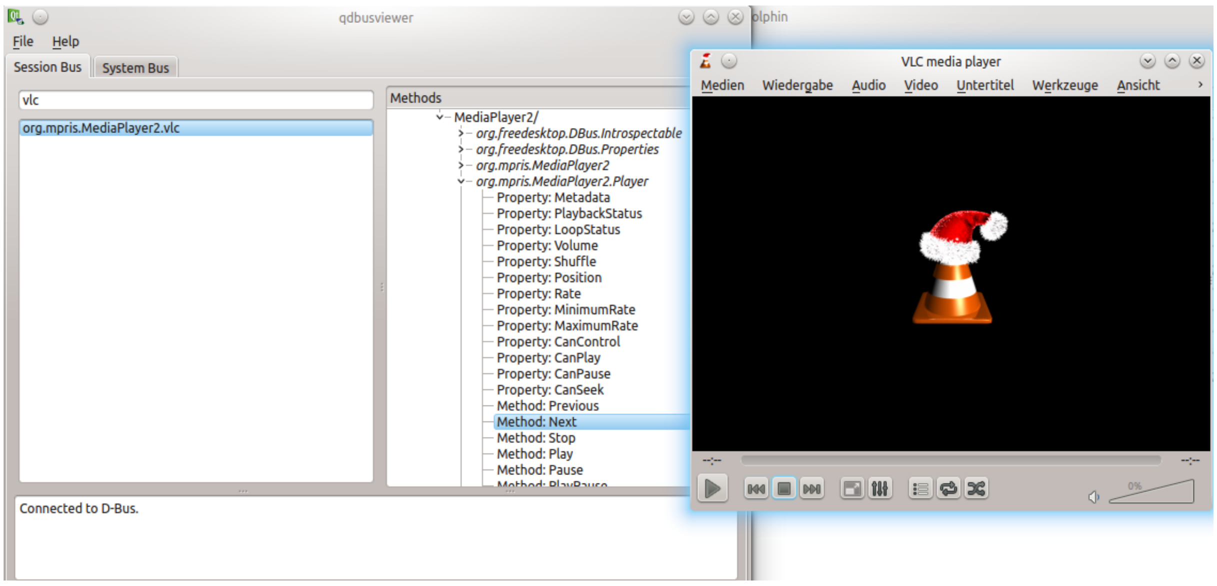
Figure 3: D-Feet on System Bus "org.freedesktop.UDisks2"

In D-feet it is also possible to execute methods supplied with its specified arguments. Return values, if any, are shown.

4.3 qdbusviewer

Another helpful tool to visualise D-Bus services is qdbusviewer which is packaged with the qt4-dev-tools. With qdbusviewer another feature is available: Connecting to signals. Once connected, for instance to signal `ActiveChanged` of interface `"org.freedesktop.ScreenSaver"`, the connection enables qdbusviewer to send its changed signal which can be of interest to other applications. [see Marg11, 14 f.]

In figure 4 the application VLC becomes introspected on bus `"org.mpris.MediaPlayer2.Player"`. Available methods e.g. `Next`, `Stop`, `Play` and properties e.g. `PlaybackStatus`, `Shuffle`, `Position` are displayed.



```
vlc
org.mpris.MediaPlayer2.vlc

Methods
MediaPlayer2/
  org.freedesktop.DBus.Introspectable
  org.freedesktop.DBus.Properties
  org.mpris.MediaPlayer2
  org.mpris.MediaPlayer2.Player
    Property: Metadata
    Property: PlaybackStatus
    Property: LoopStatus
    Property: Volume
    Property: Shuffle
    Property: Position
    Property: Rate
    Property: MinimumRate
    Property: MaximumRate
    Property: CanControl
    Property: CanPlay
    Property: CanPause
    Property: CanSeek
    Method: Previous
    Method: Next
    Method: Stop
    Method: Play
    Method: Pause
    Method: PlayPause

Connected to D-Bus.
```

Figure 4: qdbusviewer on Session Bus "org.mpris.MediaPlayer2.Player"

5 Nutshell Examples

In the implementation on nutshell examples the concepts of D-Bus scripted with ooRexx shall be shown in six scripts controlling twelve applications. Every script will pursue a different purpose including working with remote service objects, invoking simple methods, listening to applications and reacting on signals as well as an own client/server concept communicating with each other. In figure 5 three different concepts are visualised.

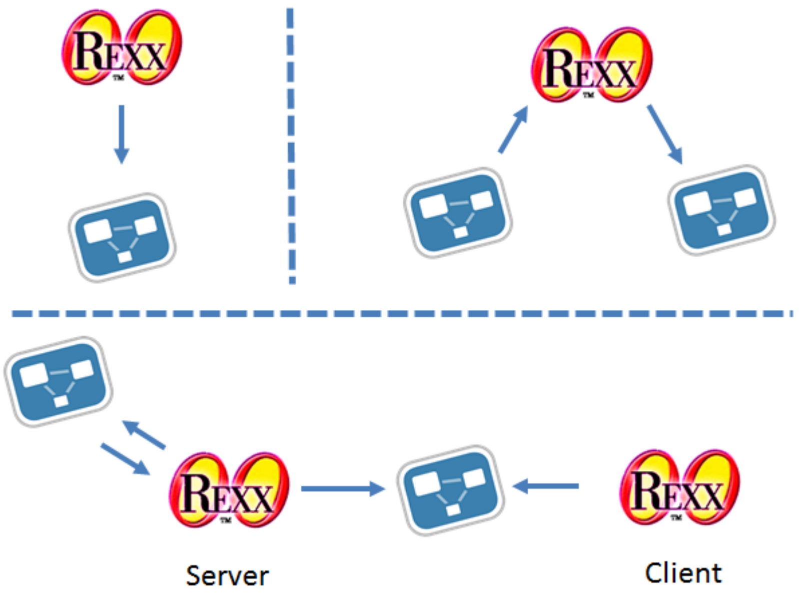


Figure 5: Concept of Nutshell Examples

- **ooRexx Script Controls D-Bus**
  Listing 4 and 6 will use service objects to send messages.
- **Listening to and Reacting on Signals**
  With the help of the listener examples 7, 8 and 9 the script will react on signals and invoke further methods.
- **Client-/Server Infrastructure**
  An own client/server infrastructure is set up with help of the method `introspect` and remote service objects. The client can access all methods provided by the server script.

5.1 A Powermode-Activity

In the example "Powermode-Activity" the script will change the KDE activity depending on the machine's power mode (battery in use).

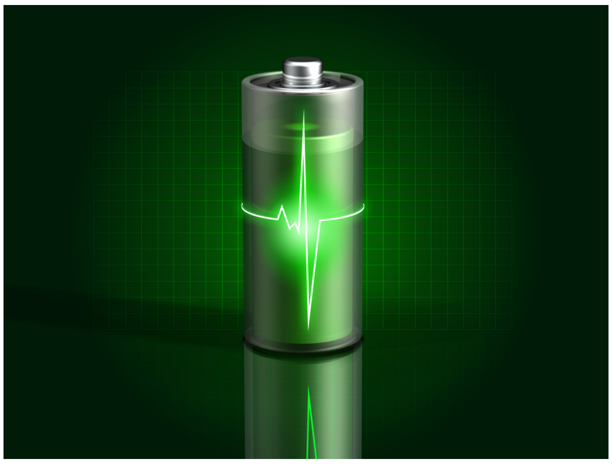


Figure 6: A Powermode-Activity

As introduced in chapter 3.1.4 two remote service objects are used, a system bus object as well as a session bus object with paths as displayed in line 2 and 3 in listing 3. The ooRexx script listing 4 will access the property `onBattery` in line 5 and 9 and change the KDE activity with the help of method `SetCurrentActivity` (with a string containing an internal reference to the KDE activity). In the activity for battery mode figure 6 is displayed as the desktop background to remember the user he or she runs on battery. If the machine runs on power supply the script changes back to the normal desktop activity.

```
 1  #!/usr/bin/rexx
 2  battery=.dbus~system~getObject("org.freedesktop.UPower","/org/freedesktop/UPower")
 3  desktop=.dbus~session~getObject("org.kde.ActivityManager","/ActivityManager/Activities")
 4
 5  if battery~OnBattery then do
 6    say "batterymode!"
 7    desktop~SetCurrentActivity("c6c608eb-110c-4e87-80f6-89b48a66f75b")
 8  end
 9  else do
10    say "power plugged!"
11    desktop~SetCurrentActivity("d5394450-b9ed-4ebc-acec-d4c96668e24c")
12  end
13
14  .dbus~system~close           -- close, thereby terminating message loop thread
15  .dbus~session~close
16  exit
17
18  ::requires "dbus.cls"         -- get access to DBus
```

Listing 4: powermode.rex

5.2 A Screenprotector

The "Screenprotector" will ask the user for a password to ensure he or she is granted to work with the computer. If password input fails, the script will lock the computer and show the screensaver.

Firstly, a short script in listing 5 will help to setup a password for the computer. In line 6 the command gets routed to Linux bash to invoke "md5sum" and save the password hash to the file "password.txt". The md5sum is a hash function tool which generates a fingerprint of an application. Remember, md5sum is compromised in security. However, for illustrating purposes the function works fine. For further information please read documentation of md5sum by issuing the command `man md5sum` or see http://wiki.ubuntuusers.de/md5sum.

```
 1  #!/usr/bin/rexx
 2
 3  say "Please define a password to protect your computer:"
 4  pull password
 5
 6  ADDRESS "bash" "echo -n "password" | md5sum > /home/richard/Dokumente/DBus/dbusoorexx/-
 7          nutshells/screenprotector_encrypt/password.txt"
 8
 9  exit
10  ::requires "dbus.cls"         -- get access to DBus
```

Listing 5: createpassword.rex

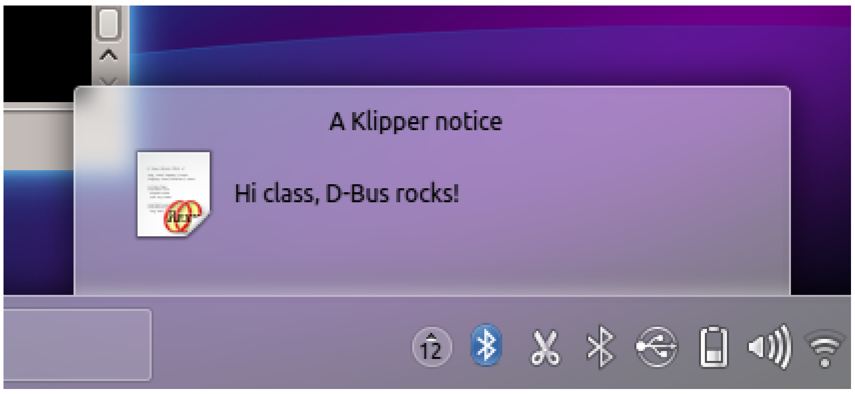
As introduced in chapter 3.1.4 a remote service session bus object is used, created in listing 6, line 2. After invoking again md5sum to verify the pulled password the method `Lock` can be invoked with no arguments. If the user typed the wrong password, the computer becomes locked.

```
 1  #!/usr/bin/rexx
 2  o=.dbus~session~getObject("org.freedesktop.ScreenSaver","/org/freedesktop/ScreenSaver")
 3
 4  say "Please enter your password: "
 5
 6  pull pw
 7  ADDRESS "bash" "echo -n "pw" | md5sum > /home/richard/Dokumente/DBus/dbusoorexx/-
 8          nutshells/screenprotector_encrypt/inputpassword.txt"
 9
10  userFile = .stream~new("password.txt")
11  userPassword = userFile~linein
12
```

```
tempFile = .stream~new("inputpassword.txt")
tempPassword = tempFile~linein

if tempPassword = userPassword then say "Correct Password!"
else o~Lock()

ADDRESS "bash" "rm /home/richard/Dokumente/DBus/dbusoorexx/-
        nutshells/screenprotector_encrypt/inputpassword.txt"

exit
::requires "dbus.cls"          -- get access to DBus
```

Listing 6: screenprotector.rex

5.3 A Klipper Listener



A Klipper notice

Hi class, D-Bus rocks!

Figure 7: A Klipper Notification

"Klipper" is a clipboard manager for the KDE interface. The "Klipper Listener" will print a notification (figure 7) once a text has been copied to Klipper. Klipper allows users of Unix-like operating systems running the KDE desktop environment to access a history of X Selections, any item of which can be reselected for pasting." [see Wiki14b] To react on the Klipper signal, the listener introduced in chapter 3.1.5 is used.

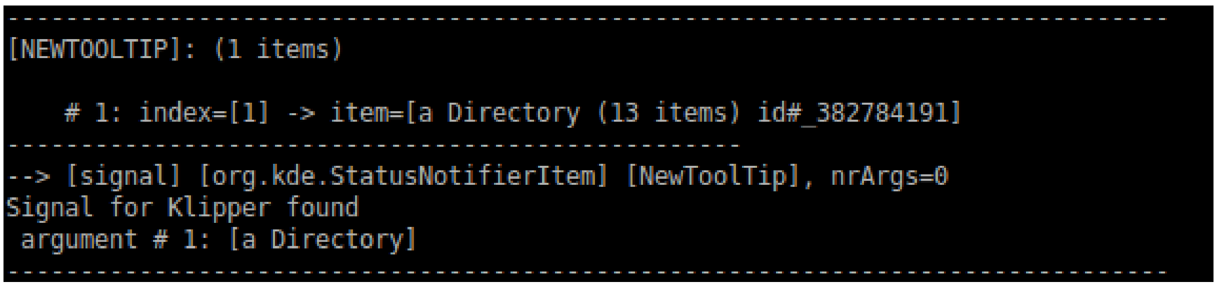
The class "RexxSignalListener" dumps all signals it receives. Therefore method unknown exists where all unknown messages become intercepted. The last argument is "slotDir" which is extracted from "methArgs"; "slotDir" is the last argument of the signal and contains e.g. the "messageTypeName", "interface" and "member" which is also prompted to command line.

In listing 7 in line 8 the listener object becomes an instance and with the help of method `match` (line 9) the script filters all messages with `type='signal'`.
Method `unknown`, line 25, intercepts all unknown messages and loops through elements

of the signals and outputs to command line as in figure 8. With the help of last argument `slotDir` properties `messageTypeName`, `interface`, `member` are accessed and shown in the terminal windows together with the number of supplied arguments (line 30). Once the `member = "NewToolTip"` for `interface = "org.kde.StatusNotifierItem"` (line 32) is found an additional prompt `"Signal for Klipper found"` is given (line 33) and the clipboard content gets accessed with method `getClipboardContents` (line 34).

A typical notification as introduced in chapter 3.1.4 will be shown containing the content of clipboard.

The routine pp will enclose the values with brackets `[...]` to be pretty readable on the command line.



```
------------------------------------------------------------------------------
[NEWTOOLTIP]: (1 items)

   # 1: index=[1] -> item=[a Directory (13 items) id#_382784191]
-------------------------------------------------
--> [signal] [org.kde.StatusNotifierItem] [NewToolTip], nrArgs=0
Signal for Klipper found
 argument # 1: [a Directory]
------------------------------------------------------------------------------
```

Figure 8: A Klipper Signal

```
1  #!/usr/bin/rexx
2  signal on halt /* intercept ctl-c (jump to label 'halt:' below) */
3
4  conn=.dbus~session /* get the "session" connection */
5
6  conn~listener("add", .rexxSignalListener~new) /* add the Rexx listener object */
7  conn~match("add", "type='signal'", .true) /* ask for any signal message */
8
9  say "Hit enter to stop listener..."
10
11 parse pull answer /* wait for pressing enter */
12 halt: /* a label for a halt condition (ctl-c) */
13
14 say "closing connection."
15
16 conn~close /* close connection, stops message loop */
17 .dbus~session~close
18
19 ::requires "dbus.cls" /* get dbus support for ooRexx */
20 ::requires "rgf_util2.rex" /* collection of ooRexx-utilities */
21
22 ::class RexxSignalListener /* just dump all signals/events we receive */
23 ::method unknown /* this method intercepts all unknown messages */
24   use arg methName, methArgs, kopete
```

```
  call dump2 methArgs, pp(methName)

  slotDir=methArgs[methArgs~size] /* last argument is slotDir */
  say "-->" pp(slotDir~messageTypeName) pp(slotDir~interface) -
        pp(slotDir~member)", nrArgs="methArgs~items-1
  if slotDir~interface = "org.kde.StatusNotifierItem" &-
         slotDir~member = "NewToolTip" then do
    say "Signal for Klipper found"
    copy = .dbus~session~getObject("org.kde.klipper", "/klipper")~getClipboardContents

    .dbus~session~getObject("org.freedesktop.Notifications",-
            "/org/freedesktop/Notifications")~notify("ooRexx App", , "oorexx",-
                "A Klipper notice", copy, , , -1)
  end


  do i=1 to methArgs~items
    say " argument #" i":" pp(methArgs[i])
    arg = methArgs[i]
  end


  say "-"~copies(79)

::routine pp /* "pretty print": enclose string value with square brackets */
  parse arg value
  return "["value"]"
```

Listing 7: klipperListener.rex

5.4 A Bluetooth Listener

The "Bluetooth Listener" will display a notification (figure 9) once a bluetooth device has been connected or removed.

In order to function properly a bluetooth driver and connection software have to be installed. In many cases it may be delivered with the distribution's standard software kit. Please find kdebluetooth on https://extragear.kde.org/apps/kdebluetooth/.

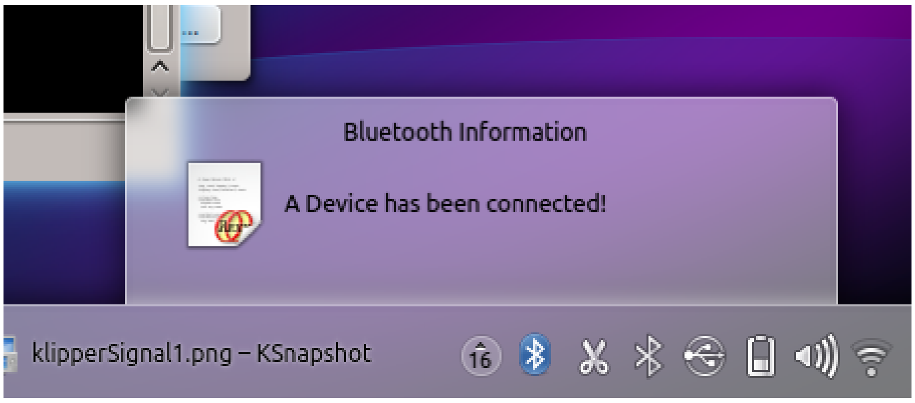
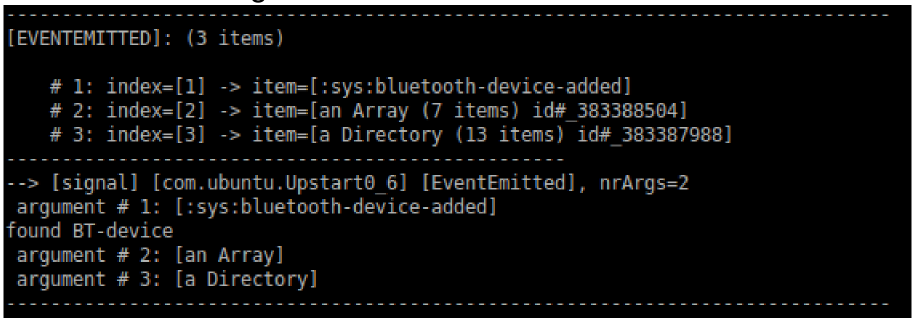


Figure 9: A Bluetooth Notification



```
--------------------------------------------------------------------------------
[EVENTEMITTED]: (3 items)

    # 1: index=[1] -> item=[:sys:bluetooth-device-added]
    # 2: index=[2] -> item=[an Array (7 items) id#_383388504]
    # 3: index=[3] -> item=[a Directory (13 items) id#_383387988]
--------------------------------------------------
--> [signal] [com.ubuntu.Upstart0_6] [EventEmitted], nrArgs=2
 argument # 1: [:sys:bluetooth-device-added]
found BT-device
 argument # 2: [an Array]
 argument # 3: [a Directory]
--------------------------------------------------------------------------------
```

Figure 10: A Bluetooth Signal

The class "RexxSignalListener" introduced in chapter 3.1.5 and used in nutshell example 5.3 will be used to print bluetooth notifications as well. Again the element `slotDir` is printed to the command line (line 28). Additionally, the bluetooth listener checks against `string ":sys:bluetooth-device-added"` and `":sys:bluetooth-device-removed"` in line 35 and 41 (figure 10) while looping through the arguments `methArgs` of the signal in line 31-46.

Once it matches, a notification figure 9 introduced in chapter 3.1.4 is displayed showing a hard-coded text `"A device has been connected"` or `"A device has been removed"`.

```
1  #!/usr/bin/rexx
2  signal on halt /* intercept ctl-c (jump to label 'halt:' below) */
3
4  conn=.dbus~session /* get the "session" connection */
5
6  conn~listener("add", .rexxSignalListener~new) /* add the Rexx listener object */
7  conn~match("add", "type='signal'", .true) /* ask for any signal message */
8
9  say "Hit enter to stop listener..."
10
11 parse pull answer /* wait for pressing enter */
12 halt: /* a label for a halt condition (ctl-c) */
13
```

```
say "closing connection."

conn~close /* close connection, stops message loop */
.dbus~session~close

::requires "dbus.cls" /* get dbus support for ooRexx */
::requires "rgf_util2.rex" /* collection of ooRexx-utilities */

::class RexxSignalListener /* just dump all signals/events we receive */
::method unknown /* this method intercepts all unknown messages */
  use arg methName, methArgs, kopete
  call dump2 methArgs, pp(methName)

  slotDir=methArgs[methArgs~size] /* last argument is slotDir */
  say "-->" pp(slotDir~messageTypeName) pp(slotDir~interface) -
  pp(slotDir~member)", nrArgs="methArgs~items-1

  do i=1 to methArgs~items
    say " argument #" i":" pp(methArgs[i])
    arg = methArgs[i]
    if arg = ":sys:bluetooth-device-added" then do
      say "found BT-device"
                .dbus~session~getObject("org.freedesktop.Notifications",-
                "/org/freedesktop/Notifications")~notify("ooRexx App", , "oorexx",-
                        "Bluetooth Information", "A Device has been connected!", , , -1)
    end
    else if arg = ":sys:bluetooth-device-removed" then do
          say "removed BT-device"
                .dbus~session~getObject("org.freedesktop.Notifications",-
                        "/org/freedesktop/Notifications")~notify("ooRexx App", , "oorexx",-
                        "Bluetooth Information", "A Device has been removed!", , , -1)
    end
  end


say "-"~copies(79)

::routine pp /* "pretty print": enclose string value with square brackets */
  parse arg value
  return "["value"]"
```

Listing 8: bluetoothListener.rex

5.5 A Kopete Listener

"Kopete" is an instant messenger supporting numerous protocols (e.g. ICQ, Yahoo, Windows Live Messenger and Skype). It is designed to be a flexible and extensible multi-protocol system suitable for personal and enterprise use. Please see https://kopete.kde.org/ for further information and the download.

The second demonstrated application is the "VideoLAN Player", "VLC". VLC is a free and open source cross-platform multimedia player and framework that plays most multimedia files as well as DVDs, Audio CDs, VCDs, and various streaming protocols.

This nutshell reacts on the VLC's playing-status in order to change the user's instant messenger status in Kopete. If he or she watches a film, Kopete will be set to offline while the user is busy.

With the help of the "Kopete Listener" the user's Kopete status will be set to offline when he or she starts to play a file in VideoLAN Player immediately. If the status of the VideoLAN Player changes back to `"Stopped"`, the user's Kopete status will be set back to online. Kopete is an instant messenger supporting different protocols (e.g. ICQ, Yahoo, Windows Live Messenger and Skype)

The script (listing 9) reacts on the signal `PropertiesChanged` when the status of the VideoLAN Player changes to `"Playing"` or `"Stopped"`. As the values are stored in a dictionary, the listener introduced in chapter 3.1.5 works through the `dict` type and checks against the status `"Playing"` or `"Stopped"`. (Cf. chapter 2.3 which describes types of D-Bus arguments.)

Once a status has changed to those predefined values, the Kopete methods `suspend` or `resume` will be invoked. (Figure 11)

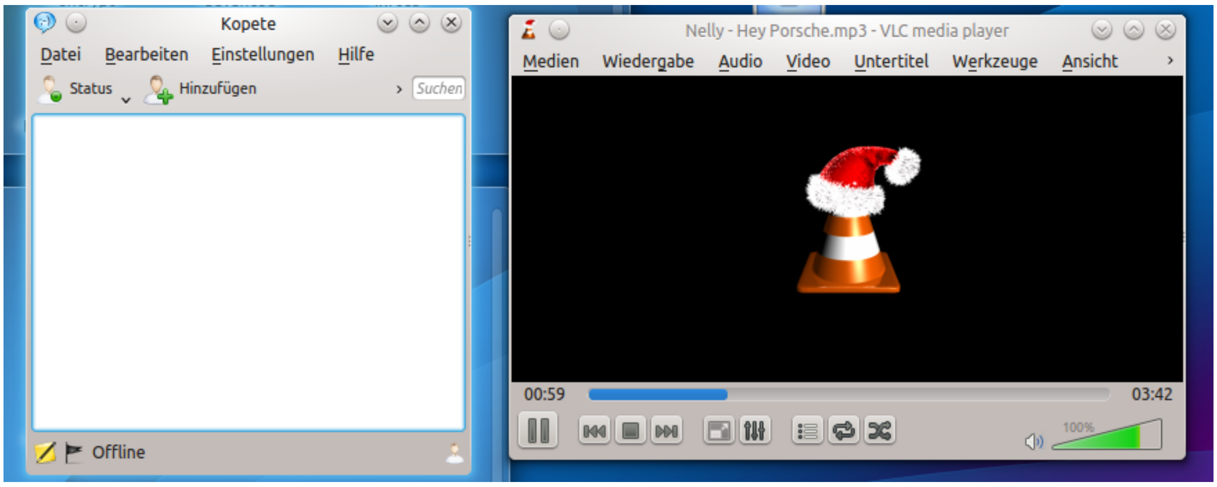


Figure 11: A Kopete Action on VLC Signal

```
#!/usr/bin/rexx
signal on halt /* intercept ctl-c (jump to label 'halt:' below) */

conn=.dbus~session /* get the "session" connection */

conn~listener("add", .rexxSignalListener~new) /* add the Rexx listener object */
conn~match("add", "type='signal'", .true) /* ask for any signal message */

say "Hit enter to stop listener..."

parse pull answer /* wait for pressing enter */
halt: /* a label for a halt condition (ctl-c) */

say "closing connection."

conn~close /* close connection, stops message loop */
.dbus~session~close

::requires "dbus.cls" /* get dbus support for ooRexx */
::requires "rgf_util2.rex" /* collection of ooRexx-utilities */

::class RexxSignalListener /* just dump all signals/events we receive */
::method unknown /* this method intercepts all unknown messages */
  use arg methName, methArgs, kopete
  call dump2 methArgs, pp(methName)

  slotDir=methArgs[methArgs~size] /* last argument is slotDir */
  say "-->" pp(slotDir~messageTypeName) pp(slotDir~interface) -
  pp(slotDir~member)", nrArgs="methArgs~items-1


  do i=1 to methArgs~items
    say " argument #" i":" pp(methArgs[i])
    arg = methArgs[i]

    if arg~isA(.collection) then
          do j over arg
          if arg~at(j) = "Playing" then do
            say "found playing VLC"
            .dbus~session~getObject("org.kde.kopete","/Kopete")~suspend
          end
          else if arg~at(j) = "Stopped" then do
            say "found paused VLC"
            .dbus~session~getObject("org.kde.kopete","/Kopete")~resume
          end
        end
  end
```

```



say "-"~copies(79)

::routine pp /* "pretty print": enclose string value with square brackets */
  parse arg value
  return "["value"]"
```

Listing 9: kopeteListener.rex

5.6 Client-/Server-Communication

In the last nutshell example an own client/server infrastructure is set up to visualise how to implement own services which can be discovered on D-Bus. The server the own service on the session bus and the client uses provided methods.

To define an own service the application gets registered on the session bus. Object path, bus name and interface can be chosen freely but can only be assigned to one application at a time. Listing 10 is the server script and lines 5-7 set the paths. In line 15 a check whether the name is free is done and if not, the script will abort. This is necessary as the service name must be unique!

In order to setup the server an instance of `HelloRexxServer` is created as a new `serviceObject` (line 22). As class "HelloRexxServer" is not a subclass of the class DBusService, the introspect path maker has to be explicity created and added as a service object in line 25. With the help of the introspect path maker own interfaces can be published.

Therefore with the method `introspect` all available interfaces of the service can be published by providing a simple XML encoded string (line 37-58) as introduced in chapter 2.5.1.

Therefore all necessary input arguments (direction="in") and return values (direction="out") have to be described for each interface. Those interfaces can be also discovered with the help of D-Feet now!

The server script provides three different introspectable methods `age`, `wlan` and `thanks`.

- **`age`**
  Method `age` will calculate the age of the user using a predefined string as an argument.

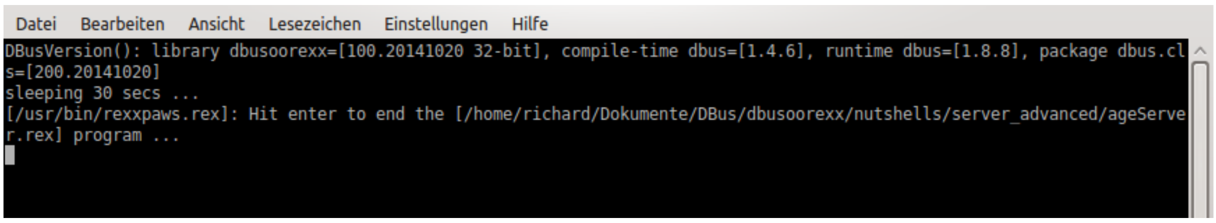
- **`wlan`**

Method `wlan` returns the status of the wireless-lan device in the computer (on/off). Therefore the property `WirelessEnabled` of `"org.freedesktop.NetworkManager"` is used.

- **`thanks`**
  Method `thanks` returns a "thank-you" string concatenated with the current date and time.

Once the server script has been started the service can be searched and used with D-Feet or other tools introduced in chapter 4.

Figure 12 displays the server script waiting for connection.

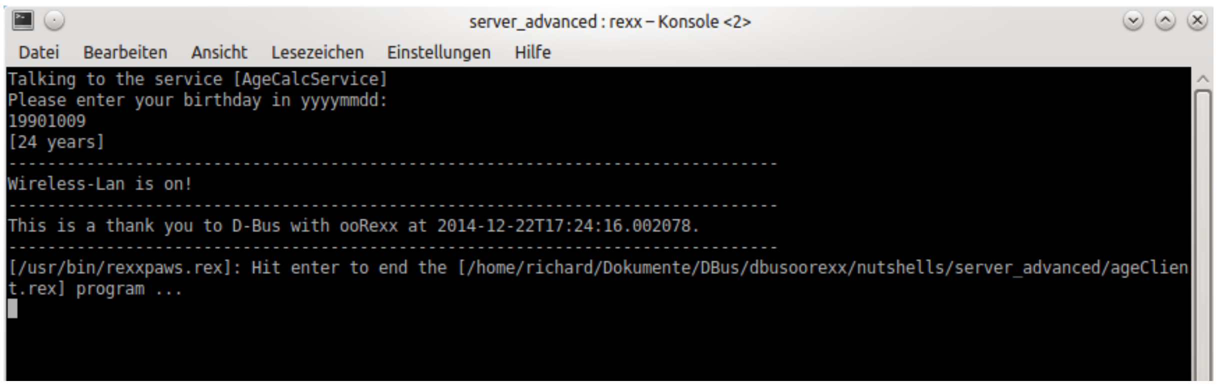


```
DBusVersion(): library dbusoorexx=[100.20141020 32-bit], compile-time dbus=[1.4.6], runtime dbus=[1.8.8], package dbus.cl
s=[200.20141020]
sleeping 30 secs ...
[/usr/bin/rexxpaws.rex]: Hit enter to end the [/home/richard/Dokumente/DBus/dbusoorexx/nutshells/server_advanced/ageServe
r.rex] program ...
```

Figure 12: A Server Listening

In the client script listing 11 it is possible to work with remote service objects as introduced in chapter 3.1.4. The script invokes the three methods in line 16-22 and the result after pulling the birthdate is displayed in figure 13.

This nutshell illustrates coverage of D-Bus services as applications can be used or monitored platform-independent over a network with full applicability of D-Bus.



```
Talking to the service [AgeCalcService]
Please enter your birthday in yyyymmdd:
19901009
[24 years]
--------------------------------------------------------------------------------
Wireless-Lan is on!
--------------------------------------------------------------------------------
This is a thank you to D-Bus with ooRexx at 2014-12-22T17:24:16.002078.
--------------------------------------------------------------------------------
[/usr/bin/rexxpaws.rex]: Hit enter to end the [/home/richard/Dokumente/DBus/dbusoorexx/nutshells/server_advanced/ageClien
t.rex] program ...
```

Figure 13: A Client Accessing

```
1
2 say "DBusVersion():" DBusVersion()
3 timeout=30    /* wait for 30 seconds (1 minute) for clients, then stop */
```

```

objectPath    ="/org/rexxla/oorexx/demo/AgeCalc"
busName       ="org.rexxla.oorexx.demo.AgeCalc"
interface     ="org.rexxla.oorexx.demo.AgeCalc"

conn=.dbus~session         /* get the session bus */

signal on syntax name halt /* make sure message loop gets stopped */
signal on halt             /* intercept ctl-c or closing terminal in which Rexx runs */

res=conn~busName("request", busName)
if res<>.dbus.dir~primaryOwner & res<>.dbus.dir~alreadyOwner then /* wait for clients*/
do
   say "res="res "problem with requesting the bus name" busName", aborting ..."
   exit -1
end

/* necessary for DBus debuggers else services are not visible */
 conn~serviceObject("add", objectPath, .HelloRexxServer~new)
 /* as class HelloRexxServer is not a subclass of the class DBusService,-
        explicitly create and store the introspect path maker */
 conn~serviceObject("Add", "default", .IDBusPathMaker~new(objectPath))

say "sleeping" timeout "secs ..."
call syssleep timeout

halt:
   conn~close           /* close, thereby terminating message loop thread */

::requires "dbus.cls"         /* get dbus support for ooRexx */

::class HelloRexxServer

::method Introspect        /* return the introspection data for this service object */
 return '<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"'  -
  '"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">    '  -
  '<node>                                                           '  -
  '  <interface name="org.freedesktop.DBus.Introspectable">         '  -
  '    <method name="Introspect">                                   '  -
  '      <arg name="data" direction="out" type="s"/>                '  -
  '    </method>                                                    '  -
  '  </interface>                                                   '  -
  '  <interface name="org.rexxla.oorexx.demo.AgeCalc">              '  -
  '    <method name="age">                                          '  -
  '      <arg name="time" direction="in" type="s"/>                 '  -
  '      <arg name="result" direction="out" type="s"/>              '  -
  '    </method>                                                    '  -
  '    <method name="wlan">                                         '  -
  '      <arg name="wlan" direction="out" type="s"/>          '  -
  '    </method>                                              '  -
  '    <method name="thanks">                                 '  -
  '      <arg name="thanks" direction="out" type="s"/>        '  -
  '    </method>                                              '  -
  '    <property name="ServiceName" access="read"  type="s"/> '  -
  '  </interface>                                             '  -
  '</node>                                                    '

::method serviceName       /* the name of this service object           */
   return "AgeCalcService"

::method age       /* the service method 'age' returns the age     */
  use arg date

  today = .DateTime~new
  birthDate = .DateTime~new(date('F', date, "S"))

  result = today - birthDate
  years = result~days()/365
  return years~round() "years"

::method wlan

  wlanStatus = .dbus~system~getObject("org.freedesktop.NetworkManager",-
        "/org/freedesktop/NetworkManager")~WirelessEnabled
  return wlanStatus

::method thanks      /* the service method 'thanks' returns a string
                        to be marshalled as 'ay' (an array of bytes)    */
  return "This is a thank you to D-Bus with ooRexx at" .dateTime~new"."

::method Get      /* access to our only property requested, return it */
  return self~serviceName
```

Listing 10: rexxServer.rex

```
#!/usr/bin/rexx

conn=.dbus~connect("session")    /* connect to the "session" bus */

objectPath    ="/org/rexxla/oorexx/demo/AgeCalc"
busName       ="org.rexxla.oorexx.demo.AgeCalc"
interface     ="org.rexxla.oorexx.demo.AgeCalc"

o=conn~getObject(busName, objectPath) /* get the DBus object */

say "Talking to the service" pp2(o~ServiceName)

say "Please enter your birthday in yyyymmdd:"
pull date

say pp(o~age(date))
say "--------------------------------------------"
if o~wlan() then say "Wireless-Lan is on!"
else "Wireless-Lan is off!"
say "--------------------------------------------"
say o~thanks
say "--------------------------------------------"

conn~close           /* close, thereby terminating message loop thread */

::requires "dbus.cls"         /* get dbus support for ooRexx */
::requires "rgf_util2.rex"    /* installed with the BSF4ooRexx package */
```

Listing 11: rexxClient.rex

6 Roundup and Outlook

This article introduced the ooRexx language binding for D-Bus, DBus4ooRexx, which allows to take advantage of the D-Bus messaging system for ooRexx programmers. The concepts and characteristics of D-Bus got described in order to understand the provided nutshell examples and to enable readers to try out nutshells and own ideas.

All nutshell examples got briefly introduced on its illustration idea and relevant lines of code. With the help of figures the outcome has been documented. The examples were sorted after its complexity aiming different intentions. All examples are prototypes and have only been tested in the author's specific environment.

A full comparison (e.g. with focus on performance issues) to other scripting language bindings should be conducted in order improve the ooRexx language binding in performance, if any issues exist. Additionally, new useful connections between Linux D-Bus services should be proved to show the D-Bus' potentialities.

References

[DbusJava14] D-Bus Java Download: Index of dbus-java
http://dbus.freedesktop.org/releases/dbus-java/, accessed on 2014-12-22.

[DbusJavaDoc14] D-Bus Java Documentation: D-Bus programming in Java 1.5
http://dbus.freedesktop.org/doc/dbus-java/dbus-java/, accessed on 2014-12-22.

[DbusPython14] D-Bus-Python: Index of dbus-python
http://dbus.freedesktop.org/releases/dbus-python/, accessed on 2014-12-22.

[Dis14] Distrowatch: Distrowatch.com
http://distrowatch.com/, accessed on 2014-12-22.

[Flat11] Flatscher, Rony G.: An Introduction to the D-Bus Language Binding for ooRexx. The 2011 International Rexx Symposium, Oranjestad, Aruba, Dutch West-Indies, 2011

[FreeD14a] The Freedesktop Project: What is D-Bus?
http://www.freedesktop.org/wiki/Software/dbus/#index4h1, accessed on 2014-12-22.

[FreeD14b] The Freedesktop Project: D-Bus Overview.
https://pythonhosted.org/txdbus/dbus_overview.html, accessed on 2014-12-22.

[KdeBlue14] The KDE Extragear: kdebluetooth
https://extragear.kde.org/apps/kdebluetooth/, accessed on 2014-12-22.

[Kop14] Kopete Instant Messenger: Kopete, The KDE Instant Messenger
https://kopete.kde.org/, accessed on 2014-12-22.

[Md5sum14] Md5sum: Wiki of md5sum
http://wiki.ubuntuusers.de/md5sum, accessed on 2014-12-22.

[Marg11] Margiol, Sebastian: Scripting the Linux D-Bus with ooRexx. Seminar Paper, Institute for Management Information Systems, Vienna University of Economics and Business Administration, 2011

[SoFo14a] Sourceforge: Open Object Rexx Download.
http://sourceforge.net/projects/oorexx/files/oorexx/4.2.0/, accessed on 2014-12-22.

[SoFo14b] Sourceforge: BSF4ooRexx Download.
http://sourceforge.net/projects/bsf4oorexx/files/GA/sandbox/dbusoorexx/, accessed on 2014-12-22.

[SoFo14c] Sourceforge: BSF4ooRexx Source Code.
http://sourceforge.net/p/bsf4oorexx/code/HEAD/tree/sandbox/rgf/misc/dbusoorexx/, accessed on 2014-12-22.

[Vlc14] VLC Player: VLC Player auf VLC.de
http://www.vlc.de/, accessed on 2014-12-22.

[Wiki14] Wikipedia: Language binding.
http://en.wikipedia.org/wiki/Language_binding, accessed on 2014-12-22.

[Wiki14b] Wikipedia: Klipper.
http://en.wikipedia.org/wiki/Klipper, accessed on 2014-12-22.

[Wu14] WU Wien: Directory listing of D-Bus Windows.
http://wi.wu.ac.at/rgf/rexx/orx22/work/, accessed on 2014-12-22.

[WuRgf14] Rgf util2: Directory listing of orx20.
http://wi.wu.ac.at:8002/rgf/rexx/orx20/, accessed on 2014-12-22.