DBusoo Rexx

Short Introduction into DBus
How to connect your ooRexx class
Short Introduction into ooTest
Examples:
  * spice up a presentation
  * automatic backups on usb device
Introduction into DBus

- DBus is a powerful message-broker system.
- Features broadcasting and receiving messages, emit and receive signals, providing services and handle properties.
- Enables easy-to-use interprocess communication between different programs.
- Programs might be written in different programming languages, run on different machines or different operating systems.
- DBus is an integrated part of almost every modern Linux distribution.

It enables a programmer to programming-language independently orchestrate different programs.
DBus object types are strictly typed.
Access to DBus is realized through so called Dbus-language-bindings.
e-dbus, pybus, QTDBus, dbus-python, Java, Perl, objective-c, Ruby, Tcl, DBusooRexx
A good language-binding ..
- tries to bring DBus interaction in line with the concepts of the programming language
- enables to circumvent the strict object type definition, DBus demands.
- Should make the application of DBus functionality as natural as possible.
## DBus Object Types

### 13 different object types

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Indicator</th>
<th>ooRexx view</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>a</td>
<td>.Array</td>
</tr>
<tr>
<td>boolean</td>
<td>b</td>
<td>Rexx String</td>
</tr>
<tr>
<td>byte</td>
<td>y</td>
<td>Rexx String</td>
</tr>
<tr>
<td>double</td>
<td>d</td>
<td>Rexx String</td>
</tr>
<tr>
<td>int16</td>
<td>n</td>
<td>Rexx String</td>
</tr>
<tr>
<td>int32</td>
<td>i</td>
<td>Rexx String</td>
</tr>
<tr>
<td>int64</td>
<td>x</td>
<td>Rexx String</td>
</tr>
<tr>
<td>objpath</td>
<td>o</td>
<td>Rexx String</td>
</tr>
<tr>
<td>signature</td>
<td>g</td>
<td>Rexx String</td>
</tr>
<tr>
<td>string</td>
<td>s</td>
<td>Rexx String</td>
</tr>
<tr>
<td>uint16</td>
<td>q</td>
<td>Rexx String</td>
</tr>
<tr>
<td>uint32</td>
<td>u</td>
<td>Rexx String</td>
</tr>
<tr>
<td>uint64</td>
<td>t</td>
<td>Rexx String</td>
</tr>
<tr>
<td>variant</td>
<td>v</td>
<td>Signature dependent</td>
</tr>
<tr>
<td>structure</td>
<td>(...)</td>
<td>.Array</td>
</tr>
<tr>
<td>map/dict</td>
<td>a{s}</td>
<td>.Directory</td>
</tr>
</tbody>
</table>

### 4 containers

- **Array** – ordered list of objects
- **Variant** – container that carries the signature of the transported value
- **Struct** – contains any object type according its signature
- **Dict** – container with string as index, carries any object type

---

**DBusooRexx makes automatic translations**
Software Requirements

- DBus is most likely already running.
- ooRexx in Version 4.2 or higher

DBus ooRexx Package

- DBUS.CLS
- Linux Systems:
  - libdbusooorexx32.so (32-bit) or
  - libdbusooorexx64.so (64-bit)
- Windows Systems:
  - libexpat.dll (32-bit) or
  - expat.dll (64-bit)

Ready for programming!
Providing your ooRexx services over DBus can be realized with few easy to follow steps.

- Create your ooRexx class and define its methods and attributes.
- Provide introspection data of your class' methods and attributes.
- Establish connection to DBus, instantiate your application, connect it and announce it.
- Your application is ready to be used from any other program that connects to DBus.
Create an ooRexx class

* Simple class with two methods & attributes
  - Method Greet and LotteryNumber
  - Attributes ServiceName and Info

```rexx
::class Demoservice
::attribute ServiceName
::attribute Info
::method init
    expose ServiceName Info
    ServiceName = 'Version ...

::method Greet
    -- the service method 'Greet' welcomes the audience
    return 'Welcome to Vienna!! Welcome to the 2015 RexxSymposium'

::method LotteryNumber
    -- needs the max range as input
    -- returns a number that will not win
    use arg maxrange
    return random(1, maxrange)
```
Provide Introspection Information

Different methods to provide Introspection

- **Version 1 - method introspect**
  - Define a method called introspect and return the introspection data through it.

- **Version 2 - xml as string**
  - Define the introspection data as string and pass it over to the class DBusServiceObject.

- **Version 3 - external xml file**
  - Define the introspection data as external xml file and pass it over to DBusServiceObject.

- **Version 4 - IntrospectHelper**
  - Use the class IntrospectionHelper and define introspection data as instructions and pass it over to DBusServiceObject.
::class Demoservice1
::method Introspect
  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="rexxsymposium.oorexx.dbus.version1">
          <method name="Greet">
            <arg name="result" direction="out" type="s"/>
          </method>
          <method name="LotteryNumber">
            <arg name="maxrange" direction="in" type="i"/>
            <arg name="magicnumber" direction="out" type="i"/>
          </method>
          <signal name="Exit">
            <arg name="result" type="i"/>
          </signal>
          <property name="ServiceName" access="read" type="s"/>
          <property name="Info" access="read" type="s"/>
        </interface>
      </node>
Method Introspect - Pros and Cons

- Comes natural to a ooRexx programmer
- Handling long strings is unhandy
- Finding errors is difficult
- No tests for closed xml brackets
- No tests for irregular DBUS syntax
- If xml data is faulty, services are not available

**No automatic marshalling** - return values need to be boxed!!

```plaintext
::method LotteryNumber
  -- needs the max range as input
  -- returns a number that will not win

  use arg maxrange
  number = random(1,maxrange)
  return dbus.box('i', number)
```
::class Demoservice2 subclass DBusServiceObject
::method init -- constructor
    expose info
    idata='  <!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN" "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
    <!---freedesktop://DTD D-BUS Object Introspection 1.0//EN" -->
    <node>
        <interface name="org.freedesktop.DBus.Introspectable">
            <method name="Introspect">
                <arg name="data" direction="out" type="s"/>
            </method>
        </interface>
        <interface name="rexxsymposium.oorexx.dbus.version1">
            <method name="Greet">
                <arg name="result" direction="out" type="s"/>
            </method>
            <method name="LotteryNumber">
                <arg name="maxrange" direction="in" type="i"/>
                <arg name="magicnumber" direction="out" type="i"/>
            </method>
            <signal name="Exit">
                <arg name="result" type="s"/>
            </signal>
            <property name="ServiceName" access="read" type="s"/>
            <property name="Info" access="read" type="s"/>
        </interface>
    </node>
self~init:super(idata) -- let DBusServiceObject initialize
XML as String & DBUSServiceObject

- Subclasses DBUSServiceObject
- Automatic marshalling according to the signature
  - Handling long strings is unhandy
  - Finding errors is difficult
  - No tests for closed xml brackets
  - No tests for irregular DBUS syntax
  - If xml data is faulty, services are not available
::class Demoservice3 subclass DBusServiceObject
::method init -- constructor
    expose info
    idata='Service3.xml'
    self~init:super(idata) -- let DBusServiceObject initialize

File: Service3.xml

```xml
<!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='rexxsymposium.oorexx.dbus.version3'>
    <method name="Greet">
      <arg name="result" direction="out" type="s"/>
    </method>
    <method name="LotteryNumber">
      <arg name="maxrange" direction="in" type="i"/>
      <arg name="magicnumber" direction="out" type="i"/>
    </method>
  </interface>

  <signal name="Exit">
    <arg name="result" type="s"/>
  </signal>

  <property name="ServiceName" access="read" type="s"/>
  <property name="Info" access="readwrite" type="s"/>
</node>
```
External XML File - Pros and Cons

- Subclasses DBUSServiceObject
- Automatic marshalling according to the signature
- Cleaner, shorter code
- XML can be edited and displayed with a dedicated application.
- Good syntax highlighting & automated syntax checks

- External File needs always be available, changes on the code have to be done on both files
- Finding errors is still difficult
- No tests for irregular DBUS syntax
- If xml data is faulty, services are not available
::class Demoservice4 subclass DBusServiceObject
::attribute ServiceName
::attribute Info
::method init -- constructor
   expose ServiceName Info

ServiceName = 'Version with IntrospectHelper'
node=.IntrospectHelper~new -- create root node
   if=node~addInterface('org.freedesktop.DBus.Introspectable')
      if~addMethod('Introspect',,'s')
   if=node~addInterface('org.freedesktop.DBus.Properties')
      if~addMethod('Get',,'ss','v')
      if~addMethod('Set',,'ssv','')
   if=node~addInterface('rexxsymposium.oorexx.dbus.version4')
      if~addMethod('Greet',,'s') -- name, in & out-signature
      if~addMethod('LotteryNumber','i','i')
      if~addProperty('ServiceName','s','read')
      if~addProperty('Info','s','readwrite')
      if~addSignal('Exit')

idata=node~makeString
self~init:super(idata) -- let DBusServiceObject initialize
IntrospectHelper - Pros and Cons

- Subclasses DBUSServiceObject
- Automatic marshalling according to the signature
- Intuitive coding, very clean code
- No worries about any line of XML code
- Automatic tests of generated code
- Rexx code syntax checks
  - number of arguments, brackets closed ..

**Provides DBUS syntax checks !!**
- Error label needs to be implemented
Refinement for Error Treatment

- IntrospectHelper throws errors if syntax rules are violated.

```plaintext
signal on syntax name halt -- make sure message loop gets stopped
signal on halt -- intercept ctl-c

halt:
errormessage = (Condition('ADDITIONAL')) -- error information
if errormessage[1]==.nil then do -- emit exit signal
ds4~service.sendSignal(objectPath, interface, 'Exit', -
                      'Goodbye, thanks for starting me')
end
else say errormessage[1]

conn~close -- close, terminating message loop thread
say 'connection closed ...'
exit -1
```
Error Treatment - Example

- IntrospectHelper throws errors if syntax rules are violated.

```plaintext
... if=node~addInterface('rexxsymposium.oorexx.dbus.version4')
   if~addMethod('Test', 'w')
   if~addMethod('Test2', 'i')
   if~addMethod('Test3','anna','i')
...```

- 'in'-signature: signature [w] contains unknown typecode 'w' at position 1
- Error 6 running /rexxsymposium/Service4.rexx line 52: Unmatched "/*" or quote
- 'in'-signature: signature [anna] Missing array element type
How to connect your ooRexx class to DBUS

* Providing your ooRexx services over DBus can be realized with few easy to follow steps.

- **Create** your ooRexx class and **define** its methods and **attributes**.
- **Provide introspection data** of your class' methods and attributes.
- **Establish connection** to DBus, **instantiate** your class, **connect** it and **announce** it.
- Your application is **ready** to be used from **any** other program that connects to DBus.
Establish connection to DBus - provide Services

- Define names according to DBUS syntax rules
- Establish a connection to the session bus
- Add an instance of your class to the connection

```rexx
objectPath     = "/rexxsymposium/oorexx/dbus/version4"
busName        = "rexxsymposium.oorexx.dbus.version4"
interface      = "rexxsymposium.oorexx.dbus.version4"

conn=.dbus~session          -- get the session bus
conn~busName('request', busName)

ds=.Demoservice~new

conn~serviceObject('add', objectPath, ds)
  .IDBusPathMaker~publishAllServiceObjects(conn)

say 'Press any key to quit'
parse pull  quit
```
Making automated tests for DBusooRexx with ooTest

Part II: Introducing ooTest and provide examples
Myths: Testing the software ...

- is not necessary for own programs
- is not worth the effort
- is only useful for a single application
- is extremly time consuming
- is extremly complicated
Testing your Program

Facts: automated tests..

- can test thousands assertions in no time
- are executable in different environments
- can easily be modified
- are very useful for other persons as well
- are easy to implement
What was tested

- Test DBus functionality
  - Messages
  - Signals
- Test creating services
  - all different possibilities to provide introspection data
  - all different possibilities to manage properties
- Test calling services
  - Test accessibility of services
- Test DBus object types
  - Test marshalling of object types, in both directions
## What was tested - Examples

### Value ranges and their boundaries

<table>
<thead>
<tr>
<th>Objecttype</th>
<th>Min value</th>
<th>Max value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int16</td>
<td>-32.768</td>
<td>32.767</td>
</tr>
<tr>
<td>unit16</td>
<td>0</td>
<td>65.535</td>
</tr>
<tr>
<td>int32</td>
<td>-2.147.483.648</td>
<td>2.147.483.647</td>
</tr>
<tr>
<td>uint32</td>
<td>0</td>
<td>4.294.967.295</td>
</tr>
<tr>
<td>int64</td>
<td>-9.223.372.036.854.775.808</td>
<td>9.223.372.036.854.775.807</td>
</tr>
<tr>
<td>uint64</td>
<td>0</td>
<td>18.446.744.073.709.551.615</td>
</tr>
</tbody>
</table>

- Wrong object types
- Missing values
- Appearance of (expected) errors
## DBus and .nil Values

<table>
<thead>
<tr>
<th>DBus Object Type</th>
<th>DBusooRexx representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>empty array</td>
</tr>
<tr>
<td>boolean</td>
<td>0</td>
</tr>
<tr>
<td>byte</td>
<td>'00'x</td>
</tr>
<tr>
<td>double</td>
<td>0</td>
</tr>
<tr>
<td>int16</td>
<td>0</td>
</tr>
<tr>
<td>int32</td>
<td>0</td>
</tr>
<tr>
<td>int64</td>
<td>0</td>
</tr>
<tr>
<td>objectpath</td>
<td>/</td>
</tr>
<tr>
<td>signature</td>
<td>empty string &quot;</td>
</tr>
<tr>
<td>string</td>
<td>empty string &quot;</td>
</tr>
<tr>
<td>uint16</td>
<td>0</td>
</tr>
<tr>
<td>uint32</td>
<td>0</td>
</tr>
<tr>
<td>uint64</td>
<td>0</td>
</tr>
<tr>
<td>variant</td>
<td>empty string &quot;</td>
</tr>
<tr>
<td>structure</td>
<td>carried object types converted to safe default</td>
</tr>
<tr>
<td>map/dict</td>
<td>empty .Directory</td>
</tr>
</tbody>
</table>

In order to assess .nil values, the expected value has to be converted to the safe default value for the given object type:

```rexx
select
  when type='g' then null = ""
  when type='y' then null = "00"x
  when type='s' then null = ""
  when type='o' then null = "/"
  otherwise
    null=0
  end

self~assertEquals(null, -dbustest~ReplyObjectPath(.nil))
```
Logic is straightforward:

- Programmer expects a certain answer from a method call.
- The method call is effected.
- The expected result gets compared with the actual result of the method call.
- After all tests have been effected, ooTest sums up.

```
Addressing Mode: 64
ooRexxUnit:      2.0.0_3.2.0   ooTest: 1.0.0_4.0.0

Tests ran:        268
Assertions:       7599
Failures:         0
Errors:           0
Skipped files:    0

Test execution:   00:04:06.085926
```
Predefined methods to test function calls

Assertions:
- `assertEquals(expected, actual, [msg])`
- `assertNotEquals(expected, actual,[msg])`
- `assertNull(actual,[msg])`
- `assertNotNull(actual, [msg])`
- `assertSame(expected, actual,[msg])`
- `assertNotSame(expected, actual,[msg])`
- `assertTrue(actual,[msg])`
- `assertFalse(actual,[msg])`
- `assertFalse(actual,[msg])`
AssertEquals vs. AssertSame

Examples:

* `assertSame „ooRexx“ and „ ooRexx “`
* `assertEquals „ooRexx“ and „ ooRexx “`
* `assertSame(1.5, dbustest~Replydouble(1.5))`
* `assertEquals(1.5, dbustest~Replydouble(1.5))`
* `assertSame(1.4, dbustest~Replydouble(1.4))`
* `assertEquals(1.4, dbustest~Replydouble(1.4))`
Error Treatment

**Intentional error:** A method that returns a string was called without an argument:

TEST_DBUSOBJECTS_STRINGS_DIRECT
Class: DBUS.testGroup
File: /home/zerkop/MasterThesis/snipplets/DBUS.testGroup
Event: [SYNTAX 93.903] raised unexpectedly.
  Missing argument in method; argument 1 is required
Program: /usr/bin/OOREXXUNIT.CLS
Line: 282

- Given the syntax number, it is possible to expect this error.
  - `self~expectSyntax(93.903)` prior to the service call that produces this error.
Testimplementation - Setup

Client-Server Architecture

- Testgroup resides on the client side
  - Takes care of necessary setup and cleanup afterwards
  - Calls methods of the DBusooRexx services
  - Effect all assertions
- ooRexx Script on the server side
  - Instances multiple DBusooRexx services that provide simple reply methods
  - Informs the client upon it is ready
  - DBusooRexx services reply the object type they receive
::method setUp
- This method is always called first when the test group is executed.
- This setup requires the serverscript to be started and wait until the services are fully initialized.

::method setUp
  .local~server.ready=.false  -- set default value for "ready"
  conn=.dbus~session  -- set up a connection to the session bus
  conn~listener("add",.rexxListener~new)  -- add the Signal Listener
  conn~match("add","type='signal',interface='oorexx.dbus.ooTestService'"))
  "rexx DBUStestServer.rexx &"  -- start the external rexx program
  say "starting server"
  do while \.server.ready  -- wait until server program sends Ready
  say '.. setUp done, starting assertions'
### Listener of the Client

- Wait until Signal arrives
- Changes variable to .true
- Starts assertions

```rexx
::Class RexxListener
::method Ready -- changes the value .server.ready
  use arg text, boolean
  say 'server sent Ready signal'
  .local~server.ready = boolean -- set ready to .true
```
::method tearDown

- If all tests are executed, the method tearDown will be called automatically.
- This method is useful to reset everything
  - The serverscript is instructed to terminate all ooRexxDBus Services and closes its connection to DBus
  - The clientscript closes its DBus connections
Example: Viewer Okular

- Create a script that spices-up a presentation.
  - The viewer currently used is called Okular.

Steps:
- Look if Okluar is connected to DBus.
- Lookup its unique name (and process-id).
- Look for interesting methods, signals and properties.
- Think about how any of this can be useful.
- Think about what information can be useful for another application.

- Connect them and enjoy ooRexx' ease and your skills.
Okular can be found under the name `org.kde.okular`

The bus name reveals that multiple instances can be started simultaneously, as it has a process-id added.

Okular does not provide any signal nor any interesting property.

We only have listed methods at our disposal.
Investigate available Methods

🌟 We need some information that triggers an action in order to create interactivity.

- `currentDocument () ⇔ (String arg_0)`
- `currentPage () ⇔ (UInt32 arg_0)`
- `documentMetadata (String metadata) ⇔ (String arg_0)`
- `enableStartWithPrint () ⇔ ()`
- `goToPage (UInt32 page) ⇔ ()`
- `openDocument (String doc) ⇔ ()`
- `pages () ⇔ (UInt32 arg_0)`
- `reload () ⇔ ()`
- `slotFind () ⇔ ()`
- `slotGotoFirst () ⇔ ()`
- `slotGotoLast () ⇔ ()`
- `slotNextPage () ⇔ ()`
- `slotPreferences () ⇔ ()`
- `slotPreviousPage () ⇔ ()`
- `slotPrintPreview () ⇔ ()`
- `slotTogglePresentation () ⇔ ()`

✅ Possibly useful for triggering events

❌ Not useful during presentation
   - Document Metadata is of no interest
   - No need to open another document
   - Not useful to switch pages over DBus
Okular's unique DBus name uses a processID
- Query the processID via **shell command**
- Store this ID in the external **Rexxqueue**

```ruby
::routine getProcId  -- returns processid of current users newest instance
cmd='pgrep -n -x -u "$USER" okular | rxqueue'
proc=getProc(cmd)
return proc

getProc: procedure  -- execute the command, parse its output
parse arg cmd
proc=""
do while queued()>0
  parse pull proc
end
return proc
```
Connect to DBus and to okular

- Select a page that triggers the action & query for it

```plaintext
conn=.dbus~session  -- get the session connection
actionPage=20
okularProcId=getProcId()

busname='org.kde.okular-'okularProcId -- create unique bus name of okular
okular=conn~getObject(busname,'/okular')  -- get the okular object

do forever
   call syssleep  4
   if (okular~currentPage==actionPage) then do
      say 'page' actionPage 'reached'
      leave
   end
end

conn~close  -- closing connections, stop message loop thread
exit -1
```
Interact with Okular

 Possibilities to spice up a presentation

- Multimedia
- Open webpages
- Send Email notifications that the presentation will last longer if page 20 was not reached in time...

This example starts a preselected audiofile in vlc

```plaintext
do forever
  call syssleep 4
  if (okular~currentPage==actionPage) then do
    say 'page' actionPage 'reached, starting audio clip'
    .dbus~session~message('call','org.mpris.MediaPlayer2.vlc', -
      '/org/mpris/MediaPlayer2','org.mpris.MediaPlayer2.Player','PlayPause')
  leave
end
end```
How to make more out of this example

- As demonstrated an ooRexx (client) program is able to connect different programs.

- An ooRexx DBUS Service can be implemented that provides the combined service by itself.

```rexx
::method defineaction
use arg pagename, -
audiofile
```
How to make more out of this example

- It is possible to provide additional features, even without interfacing with okular at all.

- When a word is marked in a presentation, our service gets the information from klipper and starts a websearch (for example translation)
Interact with System Bus

Automated backup on USB device
Connects to system bus

1. Device is added
2. File is zipped
3. zipped File is copied on the device