DBus Overview

Tom Cocagne
<tom.cocagne@gmail.com>
version 1.0.1, August 2012

D-Bus is an Inter-Process Communication (IPC) and Remote Procedure Calling (RPC) mechanism specifically designed for efficient and easy-to-use communication between processes running on the same machine. It is intended to replace the amalgam of competing IPC frameworks in this domain with a single, unified protocol that is tailored specifically for meeting the needs of secure, intra-system IPC.

There are two primary use-cases for which D-Bus is designed:

- As a "system bus" for communicating between system applications and user sessions
- As a "session bus" for exhanging data between applications in a desktop environments

As the name implies, D-Bus uses a logical "Bus" over which connected applications may communicate. For ease of use, this communication takes place via a simple object model that supports both RPC and publish-subscribe mechanisms. Applications connected to the bus may query for the availability of objects, call remote methods on them, and request notification for the signals they emit.

The APIs for objects exported over D-Bus are defined by the interfaces they support. Interfaces in D-Bus contain a comprehensive list of all methods and signals supported by the interface as well as the exact type an number of arguments they require. Any number of interfaces may be simultaneously supported by objects. This feature may be used to achieve a similar effect to the inheritance mechanism which is available in many programming languages but not directly supported by D-Bus itself.

To support a dynamic execution environment and ease of integration, D-Bus includes a standard introspection mechanism for the run-time querying of object interfaces. The D-Bus bindings for many dynamic languages make use of this feature to prevent the user from needing to explicitly specify the remote interfaces being used.

Objects in D-Bus are uniquely identified through the combination of a **bus name** and **object path**. The **bus name** uniquely identifies a client application connected to the bus and the **object path** uniquely identifies the object within that application. Each application is automatically assigned a unique and effectively random bus name upon connection. Applications wishing to export objects over D-Bus can, and typically do, request ownership of additional "well known" bus names to provide a simple and consistent location for other applications to access at run time.

The bus name mechanism in D-Bus is similar to dynamic IP address allocation paired with DNS name resolution services. Newly connected D-Bus applications are assinged a random bus name (similar to random IP address allocation). To prevent potential clients from needing to somehow discover that random bus name in order to access the application's exported objects, the providing application may instead request additional "well-known" bus names to serve as aliases for its unique name (similar to

multiple DNS entries resolving to the same IP address). For example, the Gnome NetworkManager uses the bus name "org.freedesktop.NetworkManager" to export its objects. Rather than attempting to determine the unique, random name assigned to the network manager at connection time, applications wishing to make use of the network manager will may simply use the well-known name.

Key Components

Object Paths

D-Bus objects are identified within an application via their **object path**. The **object path** intentionally looks just like a standard Unix file system path. The primary difference is that the path may contain only numbers, letters, underscores, and the / character.

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.

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. Although technically optional, most D-Bus implementations automatically provide introspection support for the objects they export.

The naming convention for D-Bus interfaces is similar to that of well-known bus names. To reduce the chance of name clashes, the accepted convention is to prefix the interface name with a reversed DNS domain name. For example, the standard "Introspection" interface is "org.freedesktop.DBus.Introspectable"

The constratints on methods and signals names contained within D-Bus interfaces are similar to those employed by many popular programming languages. They must begin with a letter and may consist of only letters, numbers, and underscores.

Each method and signal explicitly defines the number and types of arguments they accept. These are encoded as D-Bus Signature strings.

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.

Table 1. Signature Encoding

Character	Code Data Type
y	8-bit unsigned integer
b	boolean value
n	16-bit signed integer
q	16-bit unsigned integer
i	32-bit signed integer
u	32-bit unsigned integer
x	64-bit signed integer
t	64-bit unsigned integer
d	double-precision floating point (IEEE 754)
s	UTF-8 string (no embedded nul characters)
o	D-Bus Object Path string
g	D-Bus Signature string
a	Array
(	Structure start
)	Structure end
v	Variant type (described below)
{	Dictionary/Map begin

`}`	Dictionary/Map end
`h`	Unix file descriptor

No spacing is allowed within signature strings. When defining signatures for multi-argument methods and signatures, the types of each argument are concatenated into a single string. For example, the signature for a method that accepts two integers followed by a string would be "iis".

Container Types

There are four container types: Structs, Arrays, Variants, and Dictionaries.

Structs

Structures are enclosed by parentheses and may contain any valid D-Bus signature. For example, `(ii)` defines a structure containing two integers and `((ii)s)` defines a structure containing a structure of two integers followed by a string. Empty structures are not permitted.

Arrays

Arrays define a list consisting of members with a fixed type. The array charater `a` must be immediately followed by the type of data in the array. This must be a **single, complete type**.

Examples

- `ai` - Array of 32-bit integers
- `a(ii)` - Array of structures
- `aai` - Array of array of integers

Variants

Variants may contain a value of any type. The marshalled value of the variant includes the D-Bus signature defining the type of data it contains.

Dictionaries

Dictionaries work in a manner similar to that of structures but are restricted to arrays of key = value pairs. The key must be a basic, non-container type and the value may be any single, complete type. Dictionaries are defined in terms of arrays using `{}` to surround the key and value types. Examples:

- `a{ss}` - string ⇒ string
- `a{is}` - 32-bit signed integer ⇒ string
- `a{s(ii)}` - string ⇒ structure containing two integers
- `a{sa{ss}}` - string ⇒ dictionary of string to string

Methods

D-Bus methods may accept any number of arguments and may return any number of values, including none. When method calls specify no return value, a "method return" message is still sent to the calling application. This allows applications using the remote API to know that the remote method invocation has completed even if no useful result is returned.

The only case in which a return message is not sent in acknowledgement to a D-Bus method call is if the "no reply expected" flag is sent as part of the method invocation. This is an optional D-Bus implementation feature that, if supported, may be used to suppress the generation of the reply message.

The signature strings for method argument lists and method return values may contain multiple types. For each argument accepted and each value returned, the type of the argument/return value is simply appended, in order, to the signature string. For example, a method accepting two unsigned 32-bit integers and returning two strings would use "uu" for the argument signature and "ss" for the return value signature. If the method accepts no arguments or returns no values, the signatures for those attributes are empty strings.

Signals

D-Bus signals provide a 1-to-many, publish-subscribe mechanism. Similar to method return values, D-Bus signals may contain an arbitrary ammount of data. Unlike methods however, signals are entirely asynchronous and may be emitted by D-Bus objects at any time.

By default, signals emitted from D-Bus objects will not be sent to any clients. To receive signals, client applications must explicitly register their interest in the specific signals emitted by D-Bus objects.

Bus Names

There are two types of bus names: **unique** and **well-known**. Unique bus names are assigned by the bus to each client connection. They begin with a *:*, and they are guaranteed never to be reused during the life of the bus. Unlike process ids which can roll-over and be reused, unique bus names are guaranteed to be truly unique.

D-Bus clients may request additional, "well-known" bus names in order to offer their services under names that are agreed upon by convention. This allows applications to easily find the offered services at a known location. For example, the Gnome Network Manager offers its services on the well-known bus name of *org.freedesktop.NetworkManager* to prevent prospective users from needing to determine the network manager's unique bus name.

Well known bus names have essentially the same naming requirements as DNS domain names (though bus names may include the _ character). As the whole point of "well-known" bus names is simplicity of resource naming, the accepted convention for avoiding unintentional name clashes is to prefix the well known bus names with a reversed DNS domain name: **org.frobozz.Zork**

D-Bus imposes a simple arbitration mechanism on the ownership of bus names. If a client requests an unused name, it is immediately granted. If the name is currently

owned, however, the client has two options for obtaining the name. It can attempt to steal ownership or it can place itself in a queue for eventual ownership.

To steal ownership, two conditions must be met. The application currently owning the name must have informed the bus that it is willing to relinquish ownership of the name and the stealing application must set the flag enabling the ownership request to steal the name from the owning application. If both of these conditions are not met, the request for the name will simply fail and return an error code stating that the name is currently unavailable.

To queue for ownership, the requesting application sets a flag in the ownership request indicating that if the name is not currently available, the request should be queued. If, at some point in the future, ownership is granted to the client, it will be informed of this fact by way of a signal.

Message Routing

Messages are routed to client connections by destination address and match rules. Destination address routing is used when a message's destination parameter contains a unique or well-known bus name. This is typically the case with method call and return messages which inherently require 1-to-1 communication. Signals, on the other hand, are broadcast messages with no specific destination. For these, client applications must register match rules in order to receive the signals they are interested in.

Although signal registration is the most common use for message matching rules, DBus message matching rules can be used to request delivery of any messages transmitted over the bus; including the method call and return messages between arbitrary bus clients. The messages delivered via match rules are always copies so it is not possible to use this mechanism to redirect messages away from their intended targets.

Message match rules are set via the *org.freedesktop.DBus.AddMatch* method and are formatted as a series of comma-separated, key=value paris contained within a single string. Excluded keys indicate wildcard matches that match every message. If all components of the match rule match a message, it will be delivered to the requesting application. The following table provides a terse description of the keys and values that may be specified in match rules. For full details, please refer to the DBus specification.

Table 2. Rule Elements

key	possible values	Description
type	*signal*, *method_call*, *method_return*, *error*	Matches specific message type
sender	Unique or well-known bus name	Matches messages sent by a particular sender
interface	Interface name	Matches messages sent to or from the

		specified interface
member	Any valid method or signal name	Matches method and signal names
path	Any valid object path	Matches messages sent to or from the specified object
path_namespace	Any valid object path	Matches messages sent to or from all objects at or below the specified path
destination	Unique or well-known bus name	Matches messages sent to the specified destination
arg[0,1,2...]	Any string	Matches messages based on the content of their arguments. Only arguments of string type may be matched.
arg[0,1,2...]path	Any string	Specialized matching for path-like arguments. Ex: arg0path=*/aa/bb* will match */*, */aa/*, */aa/bb/cc* but not */aa*, */aa/b*, or */aa/bb*
arg0namespace	Any string	Specialized matching for partial bus names. Primarily intended for monitoring NameOwnerChanged for a group of related bus names. Ex: *member="NameOwnerChanged", arg0namespace="com.foo.bar"* will match "com.foo.bar.baz" and "com.foo.bar.quux"

Bus Addresses

Bus addresses specify the connection mechanism (TCP, Unix socket, etc) and any extra information required for successful connection and authentication. The format for bus addresses is a transport name followed by a colon followed by an optional, comma-separated set of key=value pairs. Multiple addresses may be specified in the same string, separated by semicolons.

Unix Addresses

Unix addresses use the *unix* transport and support the following key/value pairs. Each of the key/value paris are mutually exclusive with the others so only one may be used.

Key	Values

`path`	Filesystem path of the socket
`tmpdir`	Temporary directory in which a randomly named socket file with a *dbus-* prefix will be created.
`abstract`	Name of an abstract unix socket

TCP Addresses

TCP addresses use the *tcp* transport and support the following key/value pairs. TCP sockets provide very poor security over an insecure network.

Key	Values
`host`	DNS name or IP address
`port`	Port number
`family`	"ipv4" or "ipv6"

Nonce-secured TCP Addresses

Nonce-secured TCP sockets provide *slightly* better security than raw TCP (though still very poor on an unsecure network). This transport requires the user to read a file and send the contents of that file over the connection immediately after connection. It provides little protection against a man-in-the-middle attack over a network but is reasonably secure for use over the local-loopback IP address on a multi-user system.

Key	Values
`host`	DNS name or IP address
`port`	Port number
`family`	"ipv4" or "ipv6"
`noncefile`	Path to file containing the nonce

Standard Interfaces

org.freedesktop.DBus.Peer

Ping()

Used to test for liveliness of a connection. The object path is ignored.

arguments		

returns		

GetMachineId()

Returns a hex-coded UUID representing the machine the process is running on. This id should be identical for all processes running on the same machine.

arguments		
returns	`s`	The hex-coded UUID

org.freedesktop.DBus.Introspectable

Introspect()

Returns a string containing an XML description of the object's supported interfaces.

arguments		
returns	`s`	The XML interface definitions

org.freedesktop.DBus.Properties

Get(interface_name, property_name)

Returns a Variant containing the requested property. The interface name may be '', in which case, the implementation will use the first interface it finds that supports a property with the given name.

arguments	`ss`	
returns	`v`	

Set(interface_name, property_name, value)

Sets the specified property. Similar to Get, the interface_name argument may be ''.

arguments	`ssv`	
returns		

GetAll(interface_name)

Return a dictionary of string property names to variant values.

arguments	`s`	
returns	`a{sv}`	

PropertiesChanged

Signal indicating property values have changed

- parameters: `sa{sv}as`
  1. Name of interface with changed properties
  2. Dictionary of string names to variant values of the changed properties
  3. Array of string names indicating properties that have changed but for which values are not conveyed.

org.freedesktop.DBus.ObjectManager

GetManagedObjects()

Returns the subordinate objects paths and their supported interfaces and properties.

arguments		
returns	`a{oa{sa{sv}}}`	

InterfacesAdded

Signal indicating interfaces have been added to an object

- parameters: `oa{sa{sv}}`
  1. Object path gaining the new interfaces
  2. Dictionary of interfaces names and their properties

InterfacesRemoved

Signal indicating interfaces have been removed from an object

- parameters: `oas`
  1. Object path loosing the interfaces
  2. List of interface names

Bus Services

RequestName(new_bus_name, flags)

Requests the specified bus name be assigned to the client connection. The flags argument and return value is described below.

arguments	`su`	flags Bit mask with the following fields • **0x1** - "Allow Replacement" flag. If specified, a subsequent request for the bus name by another application that specifies the "Replace Existing" flag will steal ownership of the bus name from the current owner. When the name is stolen, the original owning application will automatically be placed at the end of the name ownership queue unless the "Do not queue" flag is specified. • **0x2** - "Replace Existing" flag. If specified and the bus name is currently owned by an application that set the "Allow Replacement" flag the name will be stolen from the current owner and assigned to the calling client. • **0x4** - "Do not queue" flag. If specified and the bus name is currently owned by another application, the request will simply fail. Without this flag, the requesting application will be queued for eventual ownership.
returns	`u`	1. The name was successfully acquired 2. Failed to acquire ownership but queued for eventual ownership 3. Failed to acquire ownership and not queued for eventual ownership 4. The calling application already owns the bus name

ReleaseName(bus_name)

Releases ownership of the specified bus name

arguments	`s`	Bus name to release
returns	`u`	1 = Success, 2 = Name does not exist, 3 = Not owner

ListQueuedOwners(bus_name)

Lists the connections queued for ownership of a well-known bus name (unique connection names do not have queues).

arguments	`s`	Well-known bus name to query
returns	`as`	List of queued connection names

GetNameOwner(bus_name)

Returns the current owner of the well-known bus name. If the name currently has no owner, this method will return a org.freedesktop.DBus.Error.NameHasNoOwner error.

arguments	`s`	
returns	`s`	

NameHasOwner(name)

Checks if the specified currently has an owner

arguments	`s`	Bus name to query
returns	`b`	True if the name is currently in-use

NameOwnerChanged

Signals the change in ownership of a bus name. This signal may be used to detect the arrival of new bus names as well as the ownership changes for existing bus names.

- parameters: `sss`
  1. Bus name for which ownership has changed
  2. Unique connection id for the old owner or empty string for none
  3. Unique connection id for the new owner or empty string for none

NameLost

Signal sent to the specific application when it loses ownership of a bus name.

- parameters: `s`
  - The name which was lost

NameAcquired

Signal sent to the specific application when it gains ownership of a bus name

- parameters: `s`
  - The name which was acquired

ListNames()

Returns the list of all currently owned bus names.

arguments		
returns	`as`	

ListActivatableNames()

Returns list of all names that can be activated on the bus.

arguments		
returns	`as`	

StartServiceByName(name, flags)

Requests the specified service be started. The flags argument is currently not used.

arguments	`su`	The name of the service to start, unused flags
returns	`u`	1 = success, 2 = service already running

UpdateActivationEnvironment(environment)

Modifies the environment passed to services started by the bus. Some busses may disable this method for some or all users for security reasons.

arguments	`a{ss}`	Dictionary of key = value pairs
returns		

GetConnectionUnixUser(bus_name)

Returns the uid of the process owning the specified bus name. If the call fails for any reason, an error is returned.

arguments	`s`	
returns	`u`	

GetConnectionUnixProcessId(bus_name)

Returns the process id of the process owning the specified bus name. If the call fails for any reason, an error is returned

arguments	`s`	
returns	`u`	

AddMatch(match_rule)

Adds a message matching rule for the current connection. See the "Message Matching" section for details

arguments	s	
returns		

RemoveMatch(match_rule)

Removes a previously registered match rule

argumens	s	
returns		

GetId()

Returns the unique ID for the bus.

arguments		
returns	s	

Version 1.0.1
Last updated 2012-08-23 23:26:12 CDT