GNUstep DBusKit and D-Bus Programming Manual

Niels Grewe

Table of Contents

1 Introduction

The aim of this manual is to familiarise the reader with the concepts and tools necessary to successfully integrate a GNUstep application into a desktop environment based around message exchange through the D-Bus messaging bus facilities. The manual tries to give succinct explanation of the concepts involved, providing illustrative examples whenever possible.

It will be most useful to a reader who has basic working knowledge of the Objective-C programming language and the OpenStep APIs (either from the GNUstep implementation or from Apple's Cocoa). In depth knowledge of the Distributed Objects system or D-Bus is also beneficial but not required.

1.1 An IPC primer

A typical modern computer system executes multiple units of computation at the same time. Even with a single-core CPU, the operating system will constantly switch between different units of computation by employing different multitasking strategies. This approach has a number of advantages, e.g.:

- It facilitates isolation of processes from one another: A malignant process cannot easily modify the memory of other processes on the system.
- It allows privilege separation: It is not necessary that a web-browser has the same rights as a partitioning utility. Running both in different processes allows the operating system to assign different privileges to both.
- It increases modularity: You can easily change one part of the software on your computer without disturbing the other parts.
- If the computer has more than one CPU, computation can be sped up by running more than one process (or thread) in parallel.

To leverage these advantages effectively, different processes or applications need a mechanism for inter-process communication (IPC) that allows them to exchange information (and ensure synchronisation if needed).

One way to implement an IPC mechanism is by using the message passing paradigm. Entities in a message passing system communicate by exchanging messages with each other, which makes it a natural fit for object oriented languages, where the basic abstraction is the object.

The message passing paradigm is also used in Objective-C (actually Objective-C inherited it from Smalltalk), where you interact with objects by sending messages to them. E.g. the intended meaning of

```
[alice greet];
```

would be sending the `-greet` message to the `alice` object, which is referred to as the *receiver* of the message. This idiom can be quite easily extended beyond the single process case, which the NeXT did by including the *Distributed Objects* system in the OpenStep specification that GNUstep implements. The message passing paradigm is also employed by D-Bus, and we will look at the similarities and differences of both systems in the following sections.

1.2 Distributed Objects

The GNUstep Distributed Objects (DO) System is designed to go out of a programmer's way. Since ordinary (intra-process) usage Objective-C already has message passing semantics, Distributed Objects simply extends these semantics to objects in other processes.

This works by usage of the proxy design pattern. A proxy is a stand-in object that receives messages *in lieu* of another object and forwards them (most likely after processing them as it sees fit). In the case of Distributed Objects, the proxy will take the message that is being sent to the remote object, encode it a `NSInvocation` object and send a serialised version of the invocation to the remote process where it is invoked on the receiver it was initially intended for.

Establishing a connection to a remote object using DO is thus a simple three step process:

1. Look up a process that exposes ('vends', in DO parlance) an object.
2. Establish a communication channel to the process.
3. Create a proxy object to send messages to the remote object.

Afterwards, the generated proxy can be used just like any in-process object.

Task 1. involves the `NSPortNameServer` class which can be used to obtain a communication endpoint (`NSPort`) to a service with a specific name:

```
NSPort *sendPort = [[NSPortNameServer systemDefaultPortNameServer]
  portForName: @"MyService"];
```

Task 2. involves `NSPort` and `NSConnection`. While the former is concerned with the low-level details of encoding messages to a wire format, the latter manages sending messages over ports. A connection to the above `MyService` using the created `sendPort` could be obtained like this:

```
NSConnection *c = [NSConnection connectionWithReceivePort: [NSPort port]
                                                 sendPort: sendPort];
```

Task 3. is done by calling `-rootProxy` on the `NSConnection` object. This will return an instance of `NSDistantObject`: A proxy that will use `NSConnection` and `NSPort` to forward messages to the remote object.

```
id remoteObject = [c rootProxy];
```

The DO mode of operation has a few notable advantages:

- Usual message passing semantics apply.
- The native Objective-C type system is used in both processes. No type conversion is necessary.
- New objects can be vended implicitly by returning them from the root proxy. New proxies will be created automatically for them.
- DO can make intelligent decisions about the remote objects: If process *A* has vended object *O* to process *B* (yielding the proxy *P(O)*), and *B* latter vends *P(O)* to *A*, *A* will not use *P(P(O))*, but its local reference to *O*.

It goes without saying that DO is pretty useful and GNUstep uses it in many places. It drives, for example, the services architecture, the pasteboard server, or the distributed notification system. For further information about DO, please consult the Objective-C GNUstep Base Programming Manual. We will now turn our attention to the D-Bus IPC system.

1.3 D-Bus

Distributed Objects has already been part of NeXT's OpenStep Specification, which appeared in 1994 and thus predates the D-Bus IPC system for quite some time. But while DO is only useful in an Objective-C context, D-Bus was created to suit the needs of desktop environments such as KDE or GNOME, which use (among others) C or C++ as their core programming languages.

1.3.1 Message Busses

One core concept of D-Bus is that of the message bus. A standard desktop system that uses D-Bus usually has two active message buses, dubbed the *well-known buses*. One is the *system bus*, to which system-wide services connect, the other is the *session bus* which is started per user session and allows applications on the user's desktop to communicate.

The purpose of a bus, which is running as a separate process (the *dbus-daemon*), is to provide name-services to the connected applications and route messages between them.

1.3.2 Services

A process that connects to a message bus is considered to be a *service*, even if it will not expose any object to the bus. A unique name, which starts with a colon (e.g. *:1.1*) and is required for message routing, will be assigned to every service by the bus. The service can also request further names from the bus. A text editor might, for example, want to request the name *org.gnustep.TextEditor* from the bus. These names are referred to as *well-known names* and usually utilise reverse-DNS notation.

These names can be subject to different assignment policies. A service can specify that it wants to be queued for a name that has already be assigned. It will then become the owner of the name when the last previous owner exits or releases the name. Alternatively, the service can request to replace an existing name, a feature that can be used to ensure that only one application of a specific type is running (as would be the case for, e.g., a screensaver).

1.3.3 Object Paths

When using DO, the object graph vended by a service is generated implicitly: If a message send to a remote object returns another object, that object will implicitly be vended and wrapped in a proxy for use by the other process. D-Bus operates quite differently in that respect: Every object needs to be assigned a name that can be used by remote processes to interact with the object. These object names are organised in the directory-like structure, where each object is uniquely identified by its *object path*. The UDisks service (*org.freedesktop.UDisks*) on the system bus does, for example, expose different disks of a computer at different paths:

```
/org/freedesktop/UDisks/devices/sda
/org/freedesktop/UDisks/devices/sdb
```

It is worth noting that it is a D-Bus convention to have the root object of the service not reside at the root path ("/") but at one that corresponds to the service name with all dots replaced by the path separator. Thus you do not access the root object of *org.freedesktop.UDisks* at "/" but at "/org/freedesktop/UDisks". The reason for this is to ensure proper name-spacing if different code modules in a single process have registered multiple names on the bus (which will all point to the same unique name).

1.3.4 Interfaces

D-Bus object-path nodes are the receivers and senders of D-Bus messages. They receive calls to methods and emit signals, which are broadcast by the bus and can be watched for by other applications. These methods and signals can be aggregated into *interfaces*, which are a bit, but not quite, like Objective-C protocols. One interface that almost every D-Bus object implements is *org.freedesktop.Introspectable*, which has as its sole member the `Introspect()`-method. This will return XML-encoded information about all methods, signals, and properties the object exposes.

Interfaces are also used as namespaces for their members: Identically named methods with different implementations are allowed to appear in multiple interfaces, something that is not possible with Objective-C protocols.

1.3.5 Type System

For arguments and values of methods, signals, and properties, D-Bus defines its own type system, which is similar to the C type system. It contains integer and floating point types of different sizes as well as array and structure types. The type system represents dictionaries as arrays of ordered pairs. Additionally, there is a type available for references to objects (but these references are only valid within a single service) and a variant type that, just like Objective-C's `id`, allows for values of arbitrary types. This type system has to be adopted by any application that wants to interface with D-Bus.

1.4 Comparison

Feature	**Distributed Objects**	**D-Bus**
IPC paradigm	message passing	message passing
type system	native Objective-C type system	custom D-Bus type system (C-like)
supported programming languages	Objective-C[1]	many languages through bindings
polymorphism	no special provisions	through overloaded method names in different interfaces
object-graph generation	implicit	explicit with named objects
name service	provided by separate nameserver objects	integrated
delivery of broadcast information	distributed notification system implemented on top of DO	integrated as D-Bus signals

[1] Please note that the GNUstep and Apple implementations of Distributed Objects are incompatible.

2 Using D-Bus From Objective-C

In order to access D-Bus services from an Objective-C application, the DBusKit framework is required. It provides infrastructure for managing connections to D-Bus message buses and translating Objective-C message sends to D-Bus method calls. This way, DBusKit can make interacting with D-Bus objects appear quite similar to the way one usually interacts with the DO system.

2.1 Generating Protocol Declarations With dk_make_protocol

If your application wants to invoke methods on D-Bus objects, some preparations are required: As with all other code, you need to provide declarations for the methods you want to invoke. You can either do this by writing them manually or let the **dk_make_protocol** tool generate them for you. This is possible if an .interface-file containing the introspection data for the interface exists. Calling **dk_make_protocol** with the “**-i**” switch and the name of the .interface-file will generate a header file with an Objective-C protocol declaration for that interface. For the hypothetical interface file for *org.freedesktop.Introspectable*, **dk_make_protocol** might generate the following header file:

```
#import <Foundation/Foundation.h>
/*
 * Objective-C protocol declaration for the D-Bus
 * org.freedesktop.Introspectable interface.
 */
@protocol org_freedesktop_Introspectable

- (NSString*)Introspect;

@end
```

The generated header file does only contain method declarations with arguments and return values that are Objective-C classes. The following default mappings between Foundation classes and D-Bus types are defined:

NSNumber	booleans (b), integers (y, n, q, i, u, x, t), floating point values (d)
NSString	strings (s)
DKProxy	object paths (o)
NSFileHandle	file descriptors (h)[1]
NSArray	arrays (a?), structs ((?*))
NSDictionary	dictionaries (a{??})
id	variants (v)

[1] Support for passing filedescriptors requires D-Bus 1.3.1 or later.

Here "?" denotes a single complete D-Bus type signature and "*" denotes possible repetition. It is, however, possible to use the plain C types corresponding to the D-Bus types, because DBusKit is capable of determining all necessary conversions. Thus the following declarations all specify valid ways to invoke `NameHasOwner()` method from *org.freedesktop.DBus*:

```
- (NSNumber*)NameHasOwner: (NSString*)name;
- (NSNumber*)NameHasOwner: (char*)name;
- (BOOL)NameHasOwner: (NSString*)name;
- (BOOL)NameHasOwner: (char*)name;
```

By default, **dk_make_protocol** generates protocol declarations that are compliant with Objective-C 2. It will thus produce `@property`-style declarations for properties of D-Bus objects. This behaviour can be disabled by passing the "**-1**" switch to the programme.

2.2 Obtaining a Proxy to a D-Bus Object

With these provisions in place, it is quite easy to obtain a proxy to a D-Bus object. The process is quite similar to creating a proxy to a distant object using DO. First, you create the required ports:

```
DKPort *sPort = [[DKPort alloc] initWithRemote: @"org.freedesktop.DBus"
                                         onBus: DKDBusSessionBus]
DKPort *rPort = [DKPort sessionBusPort];
```

If a service on the system bus was the desired target, one could pass `DKBusSystemBus` as the second argument of the `DKPort` initialiser or use the `+systemBusPort` convenience method to create a port object without remote.

Afterwards, a connection can be obtained to the *org.freedesktop.DBus* service (which is bus itself) as follows:

```
NSConnection *c = [NSConnection connectionWithReceivePort: rPort
                                                 sendPort: sPort];
```

Please note that this is exactly the way one would create a Distributed Objects connection. Consequentially, on can obtain a proxy to an object of this service by using `-rootProxy`:

```
id remoteObject = [c rootProxy];
```

Unfortunately, a proxy to the root object of a D-Bus service is very often not useful because services tend to install their primary object at a path corresponding to the service name. DBusKit thus extends `NSConnection` with a `-proxyAtPath:` method, which can be used to obtain proxies to non-root object. It could be used to obtain a proper proxy to *org.freedesktop.DBus* like this:

```
id remoteObject = [c proxyAtPath: @"/org/freedesktop/DBus"];
```

2.3 Sending Messages to D-Bus Objects

All further interactions with the remote object are indistinguishable from interactions with an object in the local process. E.g. the introspection data of the remote object could be obtained like this:

```
NSString *introspectionData = [remoteObject Introspect];
```

2.3.1 Overloaded methods

In some cases it is, however, necessary to treat D-Bus objects special: Since D-Bus allows method names to be overloaded per interface, it might be necessary to specify which method to call. DBusKit provides two facilities to cope with this kind of situation. For one, it is possible to embed the information about the required interface in the selector string of the method to call. This is done by replacing all dots in the interface string with underscores, placing it between `_DKIf_` `_DKIfEnd_` marker and appending the method name.

Assuming a D-Bus object implements a `getBass()` method in the interfaces `org.foo.Fish` and `org.bar.Instruments`, one could distinguish between the methods by constructing the following selectors:

- `-_DKIf_org_foo_Fish_DKIfEnd_getBass`
- `-_DKIf_org_bar_Instruments_DKIfEnd_getBass`

Since this is obviously quite clumsy, it will only be feasible for simple cases.

The other facility provided by DBusKit is the `-setPrimaryDBusInterface:` method, which instructs the proxy to prefer the named interface when looking up methods. E.g. the following statements would result in a call to the correct method:

```
[remoteObject setPrimaryDBusInterface: @"org.bar.Instruments"];
id anInstrument = [remoteObject getBass];
```

2.3.2 D-Bus 'out' Arguments

Some D-Bus methods include multiple 'out' arguments (return values):

```
<method name="GetServerInformation">
  <arg name="name" type="s" direction="out"/>
  <arg name="vendor" type="s" direction="out"/>
  <arg name="version" type="s" direction="out"/>
</method>
```

For methods of this type, DBuskit will combine all values returned by the remote D-Bus object into a single `NSArray` return value. So the Objective-C method signature of the method mentioned above would be

```
- (NSArray*) GetServerInformation;
```

2.4 Accessing and changing D-Bus properties

DBusKit will automatically generate getters and setters for D-Bus properties. A D-Bus interface might, for example, specify the following property in its introspection data:

```
<property name="address" type="s" access="readwrite"/>
```

This property can then be accessed by calling `-address` and changed by calling `-setAddress:` on the proxy object. Just like with other methods, both the plain C types and the corresponding Foundation classes are valid as parameters to the getter and setter methods:

```
- (NSString*)address;
- (char*)address;
- (void)setAddress: (NSString*)address;
- (void)setAddress: (char*)address;
```

If other methods with the same names exist within the same interface of the remote object, those will take precedence over the generated getter and setter methods.

2.5 Watching D-Bus Signals

Besides responding to method calls, D-Bus objects can also actively inform remote objects about events or state changes by the use of *signals*. These signals are published to the bus and the bus will re-broadcast them to all connected entities that subscribe to the signals. DBusKit includes support for receiving D-Bus signals through the `DKNotificationCenter` class. `DKNotificationCenter` keeps to OpenStep conventions in that it delivers the signals it receives from D-Bus in the form of `NSNotification`s and is thus similar to the notification center classes provided by the Foundation library (gnustep-base).

To make use of the notification feature, it is sometimes not even necessary to create any explicit proxies. It is enough to just obtain a reference to one of the notification centers:

```
DKNotificationCenter *center = [DKNotificationCenter sessionBusCenter];
```

(Again, a reference to the notification center for the system bus can be obtained similarly by using `+systemBusCenter`.) In a very simple case, one would simply use the center to add an object as an observer of the *NameAcquired* signal from the *org.freedesktop.DBus* interface.

```
[center addObserver: myObject
           selector: @selector(didReceiveNotification:)
               name: @"DKSignal_org.freedesktop.DBus_NameAquired"
             object: nil];
```

This example also illustrates the naming convention for signals: They start with the “`DKSignal`”-identifier and continue with the interface name and the signal name separated by underscores (“`_`”). Additionally, it is possible to register a custom notification name for a signal:

```
[center registerNotificationName: @"DKNameAquired"
                        asSignal: @"NameAquired"
                     inInterface: @"org.freedesktop.DBus"];
```

If this method returns YES, it will be possible to register observers for the `DKNameAquired` notification (it might fail if the signal was already registered under another name).

Since D-Bus provides a fine-grained matching mechanism for signals, Objective-C applications can specify in great detail what kind of signal they want to receive. The full-blown version of the registration method could be called as follows:

```
[center addObserver: myObject
           selector: @selector(didReceiveNotification:)
             signal: @"NameOwnerChanged"
          interface: @"org.freedesktop.DBus"
             sender: theBus
        destination: nil
             filter: @"org.gnustep.TextEditor"
            atIndex: 0];
```

If registered as an observer this way, `myObject` would only receive a notification if a new application took ownership of the name *org.gnustep.TextEditor*.

When delivering a notification to the observer, the notification center will create a `NSNotification` with a userInfo dictionary that follows a specific format to allow the receiver to process the notification:

member	The name of the signal being emitted.
interface	The name of the interface the signal belongs to.
sender	The *unique* name of the service emitting the signal.
path	The path to the object of the service that emitted the signal.
destination	The intended receiver of the signal; might be empty if the signal was broadcast, which is usually the case.
arg0, ..., n	If the signal did specify any values to be send alongside the signal, these values will be present in keys called *arg0*, *arg1*, ..., *argn*.

Additionally, calling `-object` on the notification will return a proxy to the object that emitted the signal.

2.6 Recovering from Failure

There are two common reasons for failure when communicating with objects on D-Bus. One is that the service your application is accessing is going away. In that case, DBusKit will notify you in a way similar to Distributed Objects. This means that when the service disappears from the bus, the `DKPort` used will post a `NSPortDidBecomeInvalidNotification` to the default notification center. You can watch for this notification and attempt recovery afterwards.

A more critical reason for failure is a malfunction or restart of the D-Bus daemon. In that case, all affected ports will issue a `NSPortDidBecomeInvalidNotification` and additionally the `DKDBus` object for the bus will post a `DKBusDisconnectedNotification` with the `DKDBusBusType` identifier at the `busType` key of the userInfo dictionary. Afterwards, DBusKit will attempt to recover from the failure in the background and you cannot use D-Bus services until you receive a `DKBusReconnectedNotification`. After receiving the notification, you can perform recovery as your application requires.

Please note that usually, such recovery from bus failures will only be successful for the system bus, for which one connects to a socket address that is persistent across restarts. For the session bus the socket address is not persistent, but stored in the `DBUS_SESSION_BUS_ADDRESS` environment variable. Hence your application should assume that the user session died when it looses connection to the session bus.

2.7 Multi-Threading Considerations

By default, DBusKit runs in single-threaded mode. This means that all interaction with the D-Bus daemon happens on the runloop of the calling thread. If multiple threads try to send messages D-Bus objects, this model of execution cannot guarantee that message delivery from and to the bus daemon is successful. The framework should still be thread-safe in the sense that it will continue functioning after raising an exception due to timeouts, but the desired behaviour can only be acheived by putting DBusKit in multi-threaded mode.

In multi-threaded mode, DBusKit will exchange messages with the D-Bus daemons via a dedicated worker-thread. To enable this behaviour the `+enableWorkerThread` method must be called on `DKPort`. All processing will then take place on the worker thread. Developers should note that after doing so, it is no longer safe to call into DBusKit from `+initialize`-methods. The reason for this is that in many recent Objective-C runtimes, `+initialize` will obtain a global lock and subsequent initialisations of classes on the worker thread might cause a deadlock. Only the GNUstep Objective-C runtime (version 1.4 or later) is not subject to this limitation. Developers are encouraged to use this feature if they target recent versions of the GNUstep Objective-C runtime or do not have any code depending on using D-Bus from `+initialize`.

3 Exposing Objects On D-Bus

Unfortunately, the present version of DBusKit does not include support for exposing objects in an Objective-C application to other applications via D-Bus.

Appendix A The GNU Free Documentation License

Concept Index