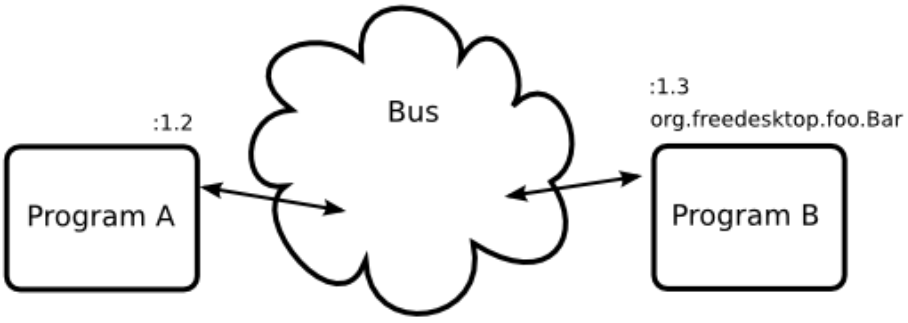
Using D-Bus

Telepathy is a D-Bus API. Telepathy components conform to the Telepathy D-Bus Specification, which is therefore also the main Telepathy API reference.

D-Bus is an IPC (Inter-process communication) system, allowing different software components running in different processes and implemented in different programming languages to communicate. D-Bus is primarily used as a server/client architecture, but one-to-one communication via a private bus is also possible. D-Bus is the defacto standard IPC mechanism for Linux.

Most of the Telepathy examples in this book will use a language binding instead of using D-Bus directly. However, an understanding of D-Bus is very helpful when learning Telepathy.

Figure 2-1 **Programs connected to a D-Bus Bus**



Message Bus

A message bus is a bus that D-Bus messages are transmitted over, brokered by a D-Bus daemon. There are two main buses that programs communicate with: the system bus (for machine wide services, e.g. HAL, NetworkManager, Avahi) and the session bus (for user/session specific services, e.g. notification messages, Telepathy, desktop session management).

Unique Name

This is an identifier assigned to a client by the D-Bus daemon (e.g. :1.3). Every client on the D-Bus has one, whether or not it is offering a named service. It is an analagous to an IP address in computer networking.

Well-Known Name

A process can make a service available by connecting to a D-Bus bus and requesting a "well-known" bus name for the connection (this is sometimes referred to as a service name), by which other processes, such as applications, can access it. If unique names are analagous to IP addresses, then well-known names are like a DNS name.

The example in Figure 2-2 provides the well-known bus name "org.freedesktop.foo.Foo".

Object Path

The service process provides D-Bus objects on that bus name. Each object has an

object path, such as "/org/freedesktop/foo/jack" (Figure 2-2), which a client application must specify to use that object.

Interface Name

Each D-Bus object implements one or more D-Bus interfaces. Each interface has an interface name, such as "org.freedesktop.foo.Jack" (Figure 2-2). Each interface provides one or more methods or signals, each with a member name.

Method

A D-Bus interface can expose a number of methods that can be called by a client. They have parameters and return types that are given as a D-Bus type signature.

Figure 2-2 gives the example of the "Fetch" method (telling Jack to fetch a pail of water).

Signal

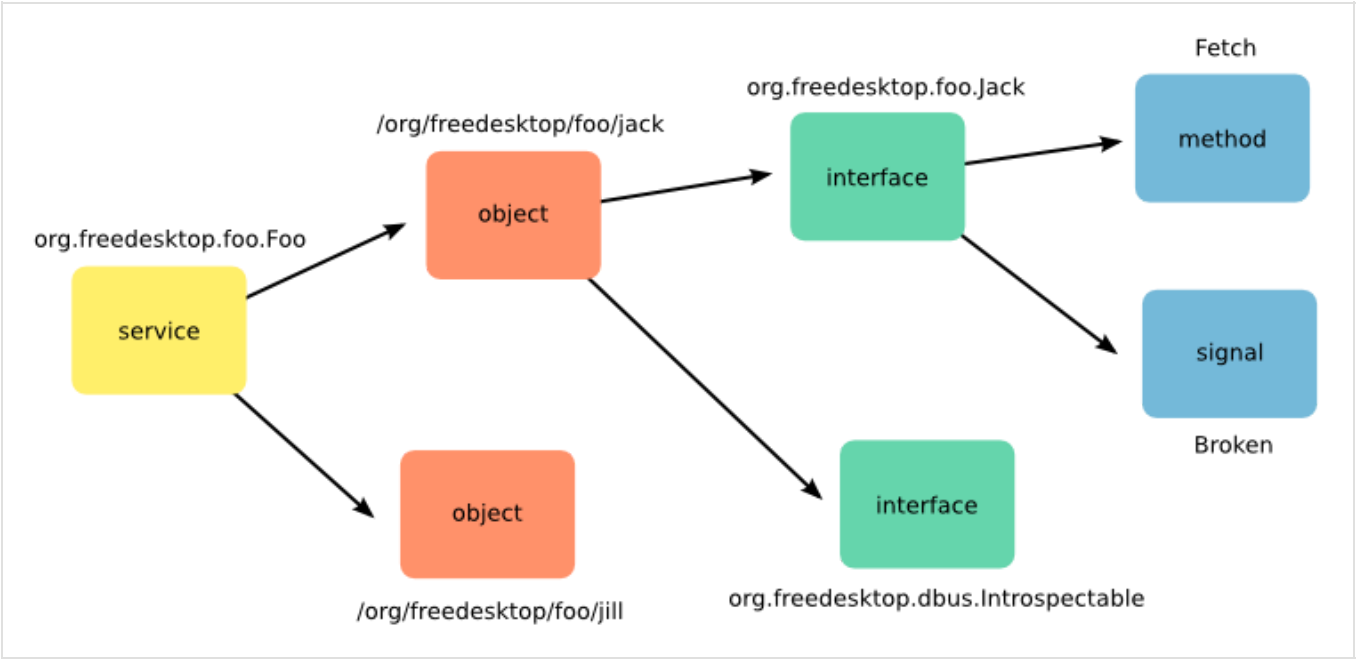
A D-Bus interface can also expose a number of signals that can be connected to by a client. Connecting a signal involves providing a callback that matches the signal's type signature that can be called by the mainloop (unlike UNIX signals, D-Bus signals are not asynchronous).

Figure 2-2 gives the example of the "Broken" signal (which is triggered when Jack falls down and breaks his crown).

Property

D-Bus objects implementing the org.freedesktop.DBus.Properties interface may also expose typed properties.

Figure 2-2 **Methods and signals on an object**



Always Avoid Synchronous D-Bus Calls

The D-Bus specification defines D-Bus as an asynchronous message-passing system, and provides no mechanism for blocking calls at the protocol level. However libdbus and most D-Bus bindings (dbus-glib, dbus-python and QtDBus) provide a "blocking" API (dbus_do_something_and_block) that implements a "pseudo-blocking" behaviour. In this mode only the D-Bus socket is polled for new I/O and any D-Bus messages that are not the reply to the original message are put on a queue for later processing once the reply has been received.

This causes several major problems:

- Messages can be reordered. Any message received before the reply and placed on the queue will be delivered to the client after the reply, which violates the ordering guarentee the D-Bus daemon provides.

  This can cause practical problems where a signal indicating an object's destruction is delayed. The client gets a method reply "UnknownMethod" and doesn't know why until the signal is delivered with more information.

- The client is completely unresponsive until the service replies (including the user interface). If the service you're calling into has locked up (this can happen, even in services that are designed to be purely non-blocking and asynchronous), the client will be unresponsive for 25 seconds until the call times out.

- The client cannot parallelize calls — if a signal causes method calls to be made, a client that uses pseudo-blocking calls can't start processing the next message until those method calls return.

- If two processes make pseudo-blocking calls on each other, a deadlock occurs.

  This sort of scenario occurs with plugin architectures and shared D-Bus connections. One plugin "knows" it's a client, not a service; and another plugin, sharing the same connection, "knows" it's a service, not a client. This results in a process that is both a service and a client (and hence deadlock-prone).

2.2.1. Naming in D-Bus

A D-Bus bus is shared with lots of other clients and services, some of which will not have been thought of yet. It is important to ensure that your well-known names, objects and interfaces all have unique names.

When choosing a well-known bus name, object name or interface name it is best practice to use a reversed domain name (as is done for Java packages) to avoid possible conflicts.

For example for well-known bus names or interfaces:

- org.freedesktop.Telepathy.ConnectionManager
- org.gnome.Project
- com.mycompany.MyProduct

For objects:

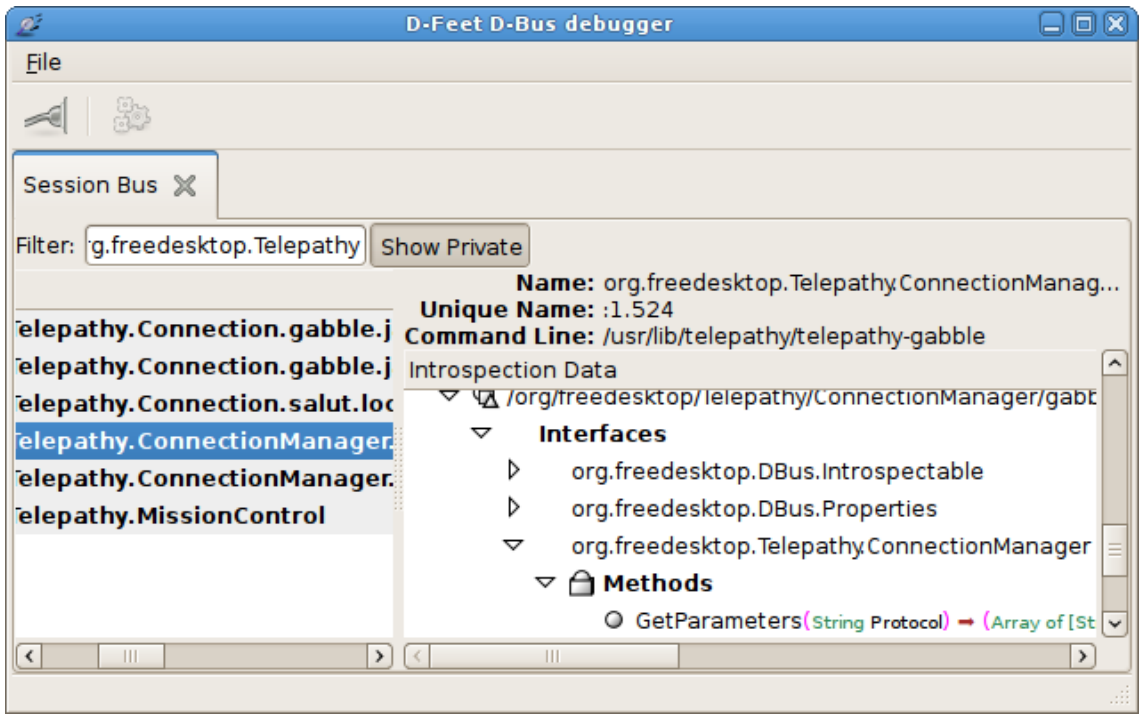
- /org/freedesktop/Telepathy/ConnectionManager/gabble
- /org/gnome/Project/adaptor
- /com/mycompany/MyProduct/object0

For simple services, with just one object that provides just one interface, these three names will often look very similar.

2.2.2. Introspecting a Bus

Many services on a D-Bus bus provide a mechanism to introspect their available objects and associated interfaces. A good utility for doing this in an interactive way is D-Feet.

Figure 2-3 **D-Feet D-Bus Introspection Tool**



D-Feet shows each service connected to the bus and the objects, interfaces, methods and signals available for that service. It allows (synchronous) method calls to be made.

2.2.3. D-Bus Type Signatures

D-Bus methods and signals are strongly typed with types given by a D-Bus type signature. The complete documentation for D-Bus type signatures is presented in the D-Bus specification.

◂ Terminology Language Bindings ▸