D-Bus with Perl

Emmanuel Rodriguez

Vienna 2008

What is D-Bus?

- A message bus system (IPC)
  - Influenced by KDE's DCOP
  - Language independent (C/C++, Python, Perl, etc)
- Simple way for applications to talk to one another
- Provides a system and session bus
- Used in Gnome, KDE, maemo, OLPC, etc

Yet another IPC?

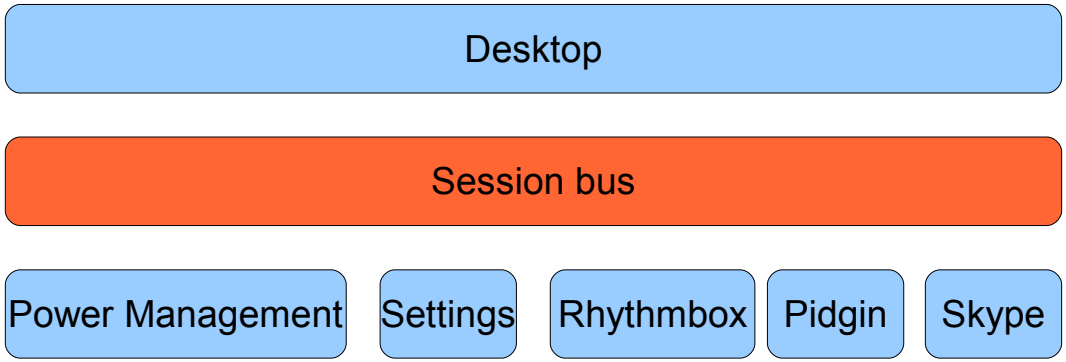
- One-to-one, peer-to-peer or client-server
- Low-latency
  - avoids round trips and can be asynchronous
- Low-overhead
  - protocol is binary and can be in machine's endianness
- Fully introspectable
- Multiple transports (Unix sockets, TCP/IP, etc)
- Supports authentication

Implementation

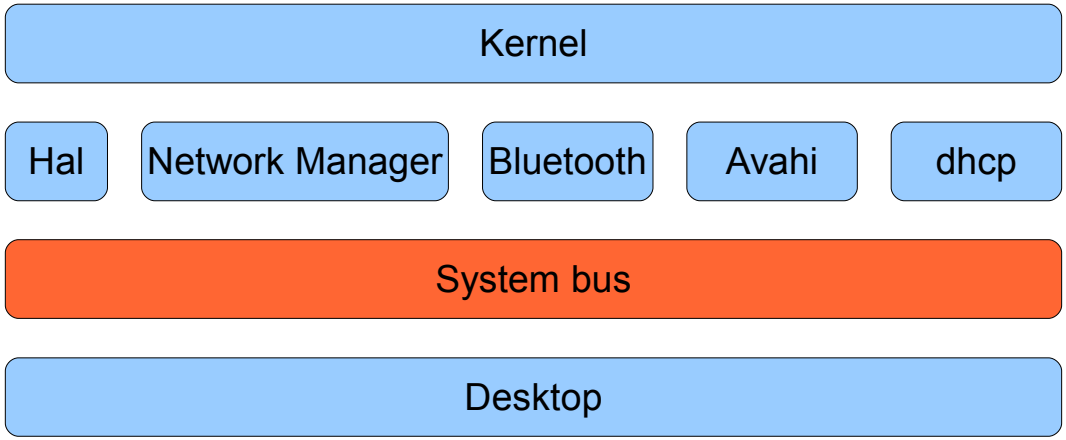
- Low level library (libdbus)
- Message bus daemon (dbus-daemon)
  - System bus for kernel events and main daemons
  - Session bus for applications and desktop session
- High level bindings
  - Glib (C)
  - Perl (Glib)
  - Python
  - Java (Glib or pure java)

Communication between

- Desktop applications in the same session
  - Programs
  - Session



- Desktop session and the OS
  - Kernel
  - System daemons



D-Bus objects

- Services (applications) are found by name
  - org.gnome.Rhythmbox
- A service can export multiple objects
  - /org/gnome/Rhythmbox/Player
  - /org/gnome/Rhythmbox/Shell
- Each object implements at least one interface
  - org.gnome.Rhythmbox.Player
  - org.freedesktop.DBus.Introspectable

Example - Client

- Connect to Pidgin and list all the accounts

```
use Net::DBus;

# Connect to Pidgin through D-Bus
my $pidgin = Net::DBus->session
    ->get_service('im.pidgin.purple.PurpleService')
    ->get_object('/im/pidgin/purple/PurpleObject')
;


# Loop through all accounts available in Pidgin
my $accounts = $pidgin->PurpleAccountsGetAll();
foreach my $account (@{ $accounts }) {

    # For each account get the protocol and the name
    my $protocol = $pidgin->PurpleAccountGetProtocolName($account);
    my $name = $pidgin->PurpleAccountGetUsername($account);

    printf "%-4s %s\n", $protocol, $name;
}
```

Example – Service (1)

- D-Bus object that can be shared

```
package DBus::Greeting;

# Invoke the exporter utility and specify the default interface name
use Net::DBus::Exporter 'org.example.Greeting';

# Become a D-Bus object
use base 'Net::DBus::Object';

sub new {
    my ($class, $service) = @_;
    # Call the parent's constructor with a service and a path for this instance
    my $self = $class->SUPER::new($service, '/org/example/Greeting');
    bless $self, $class;
}

# Register a method named “Hello” that takes a string as argument
# and returns a string
dbus_method('Hello', ['string'], ['string']);
sub Hello {
    my $self = shift;
    my ($name) = @_;
    return "Hello $name";
}
```

Example – Service (2)

- Share a D-Bus object with the world

```
use Net::DBus;
use Net::DBus::Reactor;

# Our object (previous slide)
use DBus::Greeting;

# Request a service name for the object to be shared
my $service = Net::DBus->session->export_service('org.example.Greeting');

# Export an object to our service
my $object = DBus::Greeting->new($service);

# Start a main loop and wait for incoming requests
Net::DBus::Reactor->main->run();
```

Exporting services

- The bus knows all services exported
- It can also start services on demand if a service is not running yet
- Simply create a *.service* file in /usr/share/dbus-1/services/

```
[D-BUS Service]
Name=org.example.FileWatcher
Exec=/tmp/dbus/dbus-file-watcher.pl
```

Introspectable

```
dbus-send --session --dest=org.example.FileWatcher –print-reply
          /org/example/FileWatcher org.freedesktop.DBus.Introspectable.Introspect

<node name="/org/example/FileWatcher">
  <interface name="org.example.FileWatcher">
    <method name="GetMonitoredFolders">
      <arg type="as" direction="out"/>
    </method>
    <method name="MonitorFolder">
      <arg type="s" direction="in"/>
    </method>
    <method name="Shutdown">
    </method>
    <method name="UnmonitorFolder">
      <arg type="s" direction="in"/>
    </method>
  </interface>
  <interface name="org.freedesktop.DBus.Introspectable">
    <method name="Introspect">
      <arg type="s" direction="out"/>
    </method>
  </interface>
...
</node>
```

Typical usage

- Application gets a connection to a bus
  - Each connection gets a unique bus name (:1.12)
- There's no distinction between client/service
  - A service usually:
    - Exports a service name (org.freedesktop.Hal)
    - Shares at least one object (/org/freedesktop/Hal)
    - Responds to method calls and can emits signals
  - A client usually:
    - Gets an instance to a known object (shared by a service)
    - Invokes methods or connects to signals

Messages Types

- Method Calls (client -> service)
- Responses (service -> client)
  - No reply
  - Value(s)
  - Errors (exceptions)
- Signals (service -> clients)

End

Questions?