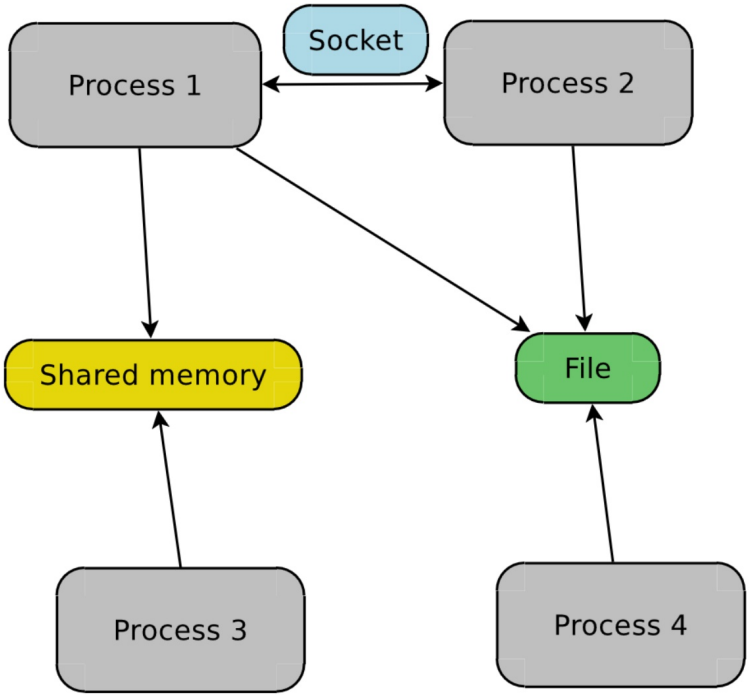
Межпроцессное взаимодействие посредством D-Bus

Артём Попцов <poptsov.artyom@gmail.com>
2024

Межпроцессное взаимодействие

- **IPC** – Inter-Process Communication
- Примеры:
  - Сигналы
  - Файлы
  - Разделяемая память
  - Сокеты
  - “Пайпы” (“Pipes”)
  - D-Bus



Сигналы (signals)

- Преимущества:
  - Сигнал легко послать от процесса к процессу и обработать.
  - Удобный способ передать простое уведомление процессу.
- Недостатки:
  - Существует ограниченный набор сигналов (обычно порядка 32-х).
  - Большинство сигналов имеют фиксированное значение.
  - Некоторые сигналы не могут быть обработаны процессом.
  - Малый объём передачи информации.

Файлы (files)

- Преимущества:
  - Позволяют передавать большой объём информации.
  - С ними относительно легко работать.
  - Являются частью файловой системы.
  - Подконтрольны общестистемным механизмам разграничения доступа.
- Недостатки:
  - Не имеют стандартизированной структуры представления информации для IPC.
  - Процессы должны знать путь к файлу.
  - Требуется контроль совместного доступа к файлу для синхронизации чтения/записи.
  - Большие накладные расходы на чтение/запись.

Разделяемая память (shared memory)

- Преимущества:
  - Быстрый способ передачи информации между процессами.
  - Потенциально большой объём памяти для передачи информации.
- Недостатки:
  - Не имеет стандартизированной структуры представления информации для IPC.
  - Процессы должны “договариваться” об совместном использовании участка памяти.
  - Требуется контроль совместного использования памяти для синхронизации чтения/записи.

Сокеты (sockets)

- Преимущества:
  - Стандартный низкоуровневый способ коммуникации процессов.
  - Позволяет взаимодействовать между процессами по сети.
  - Unix-сокеты являются частью файловой системы.
- Недостатки:
  - Передаваемые данные не имеют определённой структуры представления информации для IPC.
  - Процесс должен знать путь к сокету в файловой системе, или же IP-адрес + порт при взаимодействии по сети.

Пайпы (pipes)

- Преимущества:
  - Достаточно простой способ организации общения двух процессов.
- Недостатки:
  - Передаваемые данные не имеют определённой структуры представления информации для IPC.
  - Однонаправленная коммуникация.
  - Процессы должны быть связаны отношением родитель/потомок (ограничение обходится через именованные каналы.)

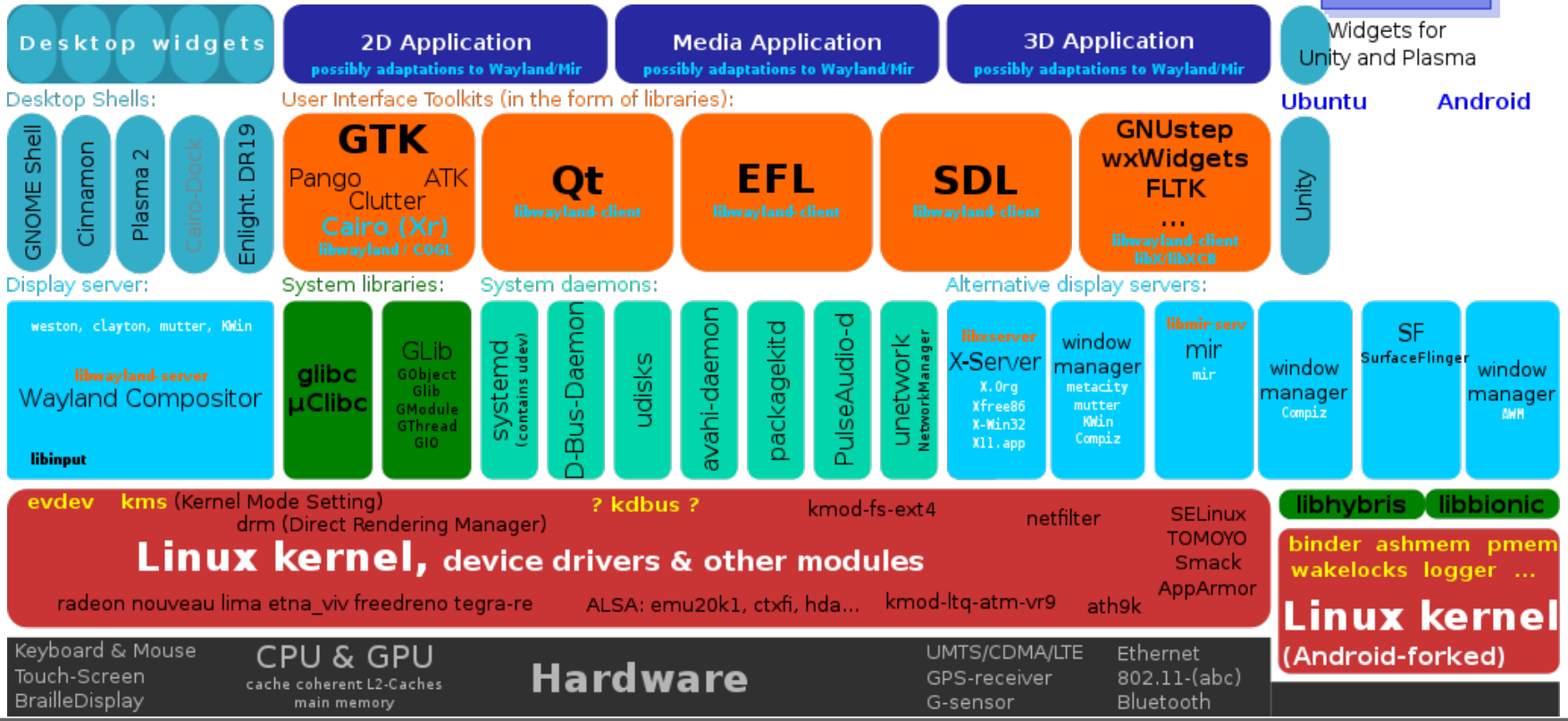
D-Bus

- Преимущества:
  - Работает “из коробки” в большинстве систем.
  - Объектно-ориентированный способ коммуникации.
  - Обмен структуированными сообщениями.
  - Удобный API, большой выбор готовых библиотек.
- Недостаки:
  - Требует запущенного D-Bus демона (сервиса.)
  - Требует специальных инструментов и библиотек.
  - Низкая эффективность передачи больших объёмов данных.

История

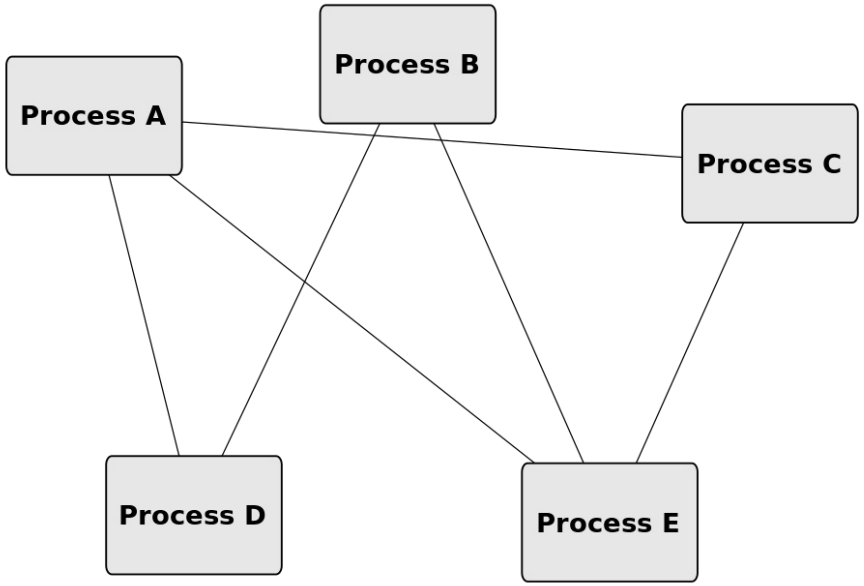
- Работа над проектом началась в 2002-м году разработчиками Havoc Pennington и Alex Larsson из компании Red Hat, и Anders Carlsson из компании CodeFactory AB в рамках проекта freedesktop.org
- Выпуск первой стабильной версии 1.0 состялся в 2006-м году.
- D-Bus пришёл на смену другим протоколам, используемым в популярных окружениях – DCOP (KDE) и Bonobo (GNOME).

Роль D-Bus в современных дистрибутивах

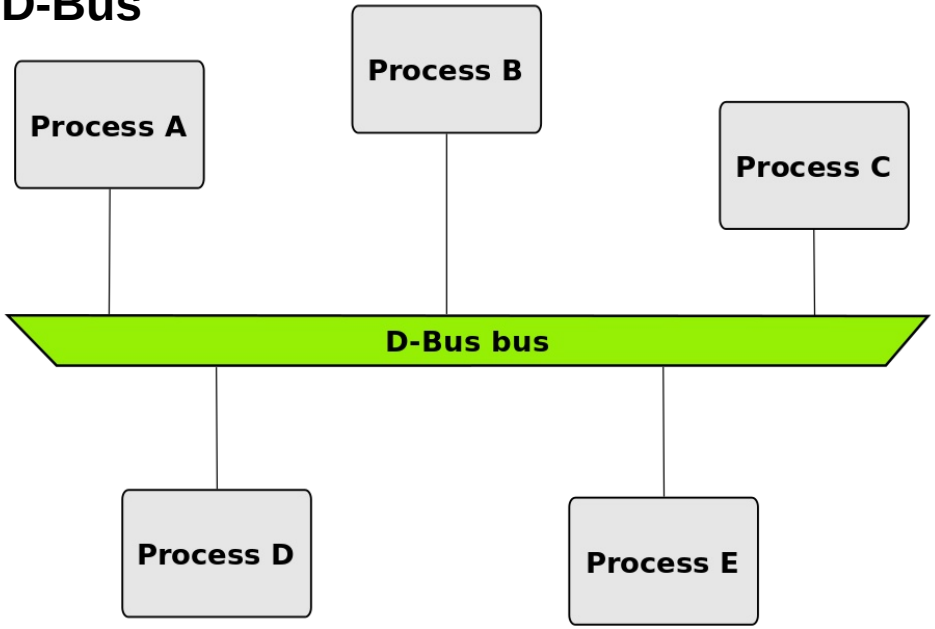


Взаимодействие между приложениями

Без D-Bus



D-Bus

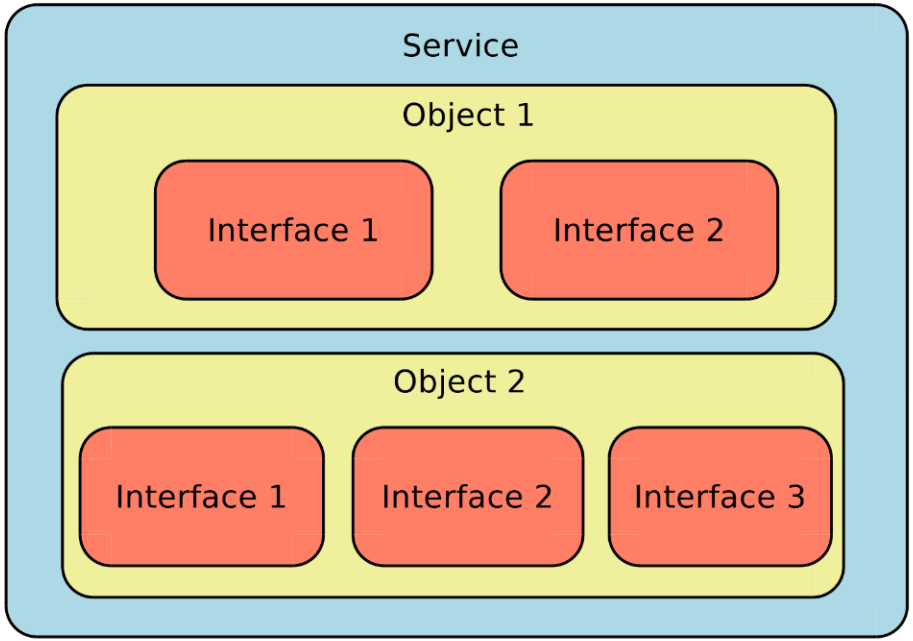


IPC на базе D-Bus

- Базируется на сокетах.
- Предоставляет слой абстракции для приложений.
- Намного проще в использовании, чем альтернативы.

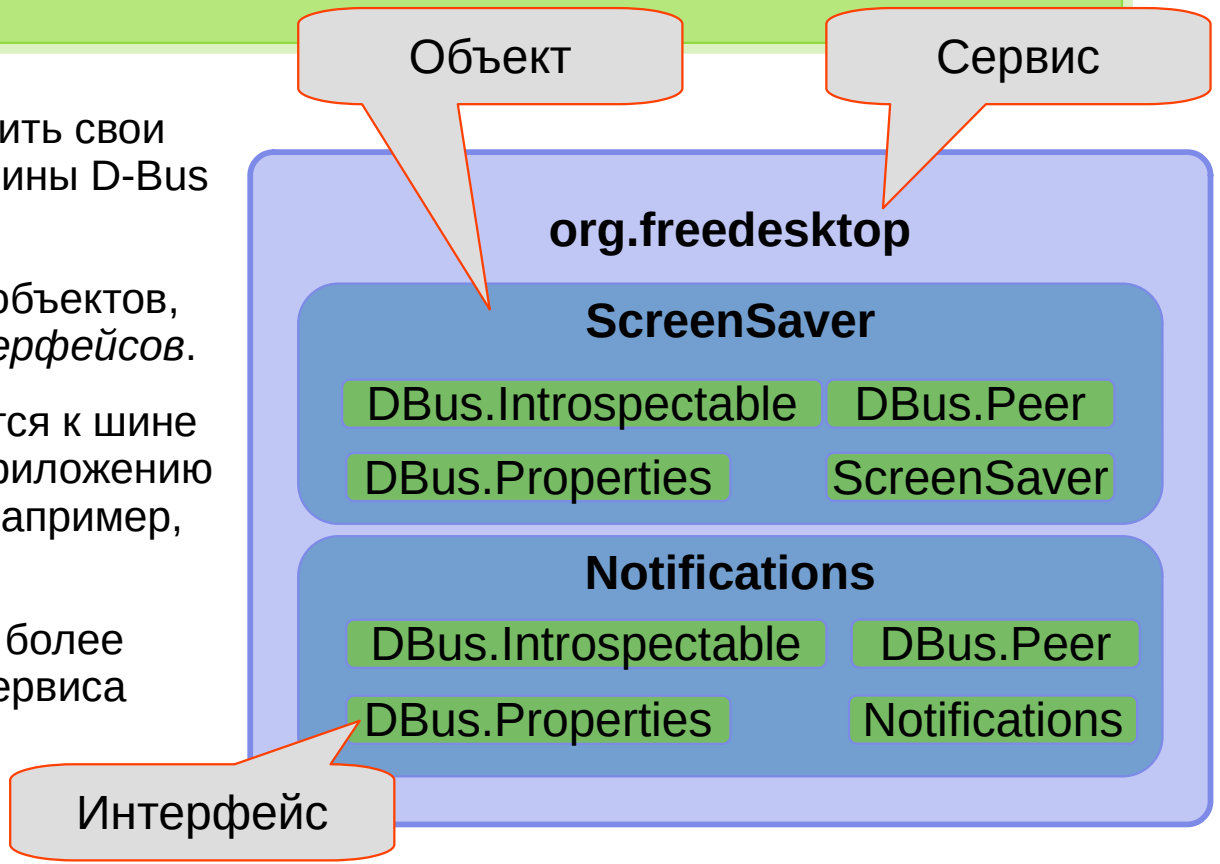
Концепции D-Bus

- Основные элементы:
  - *Сервисы* (Services)
  - *Объекты* (Objects)
  - *Интерфейсы* (Interfaces)
  - *Клиенты* (Clients) – приложения, использующие D-Bus
- Один *сервис* D-Bus содержит *объект(ы)*, которые реализуют *интерфейс(ы)*.



Сервис (Service)

- Приложение может предоставить свои сервисы для пользователей шины D-Bus путём регистрации на шине.
- *Сервис* является коллекцией объектов, предоставляющих набор *интерфейсов*.
- Когда приложение подключается к шине D-Bus, демон D-Bus выдаёт приложению уникальный идентификатор (например, “:1.40”.)
- Приложение может запросить более человеко-читаемое имя для сервиса (например, “org.freedesktop”.)



Объект (Object)

- Связан с одним сервисом.
- Может быть динамически создан или удалён.
- Уникально идентифицируется через *путь объекта*. Например: `/org/freedesktop/ScreenSaver`
- Реализует один или несколько *интерфейсов*.

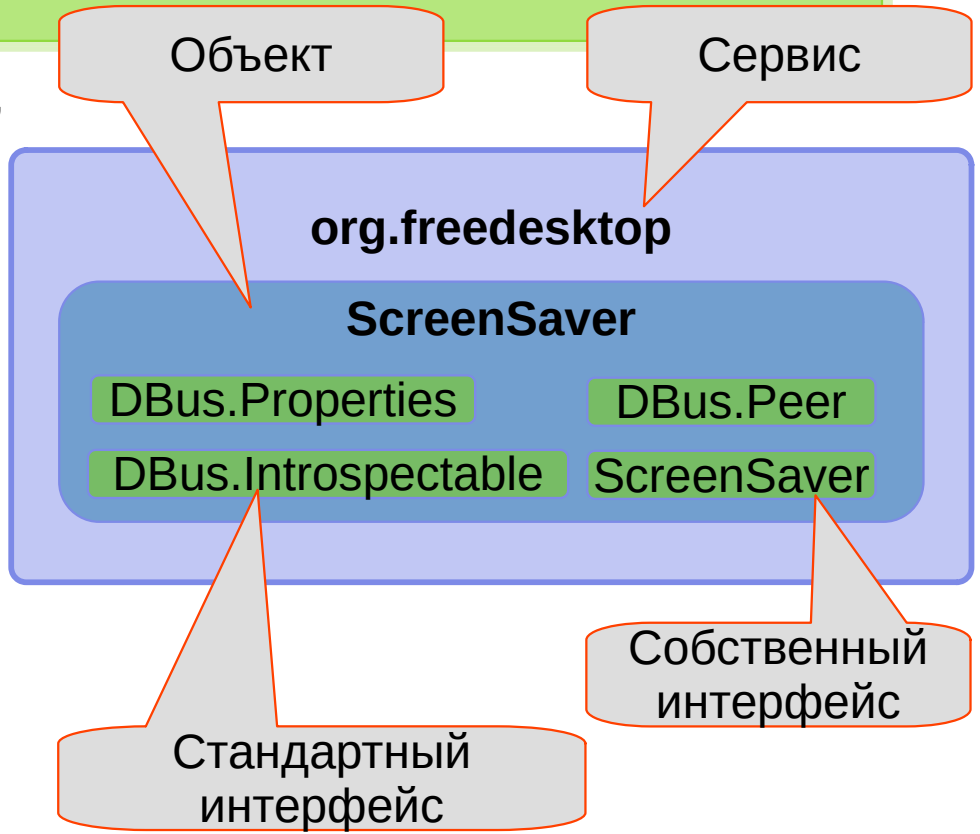
Интерфейс (Interface)

- Похож на концепцию “пространства имён” (namespace) в некоторых языках программирования (Java, C++)
- Имеет уникальное имя, компоненты которого разделяются точками (можно провести аналогию с именами классов в языке Java.)

  Например:

  `org.freedesktop.ScreenSaver`
- Содержит в себе *дочерние компоненты* (members):

  свойства (properties), *методы* (methods) и *сигналы* (signals)



Интерфейс (Interface)

- D-Bus определяет несколько стандартных интерфейсов, которые принадлежат к пространству имён **“org.freedesktop.DBus”**:
  - **org.freedesktop.DBus.Introspectable** – Предоставляет механизмы интроспекции объекта (получения информации об его интерфейсах, методах и сигналах.)
  - **org.freedesktop.DBus.Peer** – Предоставляет возможность проверять наличие подключения к объекту (“ping”.)
  - **org.freedesktop.DBus.Properties** – Предоставляет методы и сигналы для управления свойствами объекта.
  - **org.freedesktop.DBus.ObjectManager** – Предоставляет удобный API для работы с вложенными объектами.
- Интерфейсы описывают *свойства*, *методы* и *сигналы* объекта.

Свойства (Properties)

- Напрямую доступные для чтения и записи поля объекта.
- Могут иметь различные *типы* в соответствии со спецификацией D-Bus:
  - Базовые типы: **`byte`**, **`boolean`**, **`integer`**, **`double`**, …
  - “Строкоподобные” типы: **`string`**, **`object path`** (должен быть корректным), **`signature`**
  - Контейнеры: **`structure`**, **`array`**, **`variant`** (сложный тип) и **`dictionary`** (словарь)
- Имеют удобный стандартный интерфейс для работы с ними: `org.freedesktop.Dbus.Properties`

Типы данных D-Bus

- Типы имеют краткие названия, состоящие из последовательности символов.

byte	y	string	s	variant	v
boolean	b	object-path	o	array of int32	ai
int32	i	array	a	array of an array of int32	aai
uint32	u	struct	()	array of a struct with 2 int32 fields	a(ii)
double	d	dict	{}	dict of string and int32	{si}

Методы (Methods)

- Позволяют производить удалённые вызовы процедур (RPC – Remote Procedure Calls) от одного процесса к другому.
- Могут принимать параметры при вызове.
- Могут возвращать значения или сложные объекты.
- Вызов метода в D-Bus похож на вызовы обычных процедур (функций, методов) в знакомых вам языках программирования.

```
org.freedesktop.DBus.Properties :
    Get (String interface_name, String property_name) => Variant value
    GetAll (String interface_name) => Dict of {String, Variant} props
    Set (String interface_name, String property_name, Variant value)
```

Сигналы (Signals)

- Короткие сообщения / уведомления.
- Ненаправленные.
- Отправляются всем клиентам, которые “слушают” данный сигнал.
- Могут иметь параметры.
- Клиент может подписаться на сигналы для уведомлений:

```
org.freedesktop.DBus.Properties :
    PropertiesChanged (String, Dict of {String, Variant}, Array of String)
```

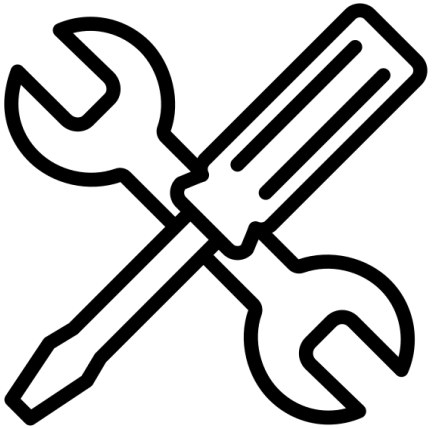
Политика (Policy)

- Добавляет механизм обеспечения безопасности.
- Представлены XML-файлами.
- Обрабатываются каждым D-Bus демоном
  Стандартные каталоги конфигураций:
  - `/etc/dbus-1/system.d`
  - `/etc/dbus-1/session.d`
- Позволяет администратору контролировать, какой пользователь к какому интерфейсу имеет доступ и т.п.
- Если вы не можете получить доступ к сервису D-Bus, или же получаете ошибку “org.freedesktop.Dbus.Error.AccessDenied”, проверьте настройки политик D-Bus!
- Сервис **“org.freedesktop.PolicyKit1”** был создан специально, чтобы позволить непривелигированным процессам контролируемо выполнять действия, требующие прав администратора.

Пример политики (/etc/dbus-1/system.d/org.freedesktop.ModemManager1.conf)

```
<!DOCTYPE busconfig PUBLIC
 "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
  <policy context="default">
    <deny send_destination="org.freedesktop.ModemManager1"
          send_type="method_call"/>
    <!-- ... -->
    <allow send_destination="org.freedesktop.ModemManager1"
           send_interface="org.freedesktop.ModemManager1"
           send_member="ScanDevices"/>
    <!-- ... -->
  </policy>
  <policy user="root">
    <allow own="org.freedesktop.ModemManager1"/>
    <allow send_destination="org.freedesktop.ModemManager1"/>
  </policy>
</busconfig>
```

- Всем по-умолчанию запрещён вызов методов на объекте **“org.freedesktop.ModemManager1”**.
- Всем по-умолчанию разрешён вызов метода **“ScanDevices”**, предоставляемый интерфейсом **“org.freedesktop.ModemManager1”**.
- Пользователь **“root”** имеет полный доступ к объекту **“org.freedesktop.ModemManager1”**.



Инструменты и библиотеки

Библиотеки

- **libdbus** – низкоуровневая библиотека.
  - *“Если вы используете API этой библиотеки напрямую, то вы подписались на великую боль.”* – с официальной страницы проекта.
- **GDbus**
  - Часть GLib (GIO)
  - Предоставляет удобный API.
- **QtDbus**
  - Модуль для Qt.
  - Удобно использовать, если у вас уже есть Qt в системе.
  - Содержит много классов для высокоуровневого взаимодействия с D-Bus.

Обёртки для разных языков

- Существуют обёртки для большинства популярных языков программирования: dbus-python, dbus-java, …
- Все обёртки позволяют:
  - Взаимодействовать с существующими D-Bus сервисами.
  - Создавать собственные сервисы, объекты, интерфейсы и т.п.
    - ...однако стоит учитывать, что D-Bus не является высокоэффективным способом передавать данные.
  - D-Bus должен использоваться для управления, а не для передачи данных.
    - Например, D-Bus может использоваться для настройки звуковой подсистемы, но не должен использоваться для передачи аудио-потока.

Компоненты D-Bus

- “libdbus” – низкоуровневая библиотека.
- “dbus-daemon” – системный сервис (демон), основанный на “libdbus”. Управляет и обрабатывает передачу данных между пирами.
- Два типа шин: “системная” и “сессионная” (подробнее – на следующем слайде.)
- Механизм обеспечения безопасности на основе конфигурационных файлов “policy”.

Виды шин

- **Системная**
  - Создаётся при старте демона D-Bus. С её помощью происходит общение различных демонов, таких как UPower, а также взаимодействие пользовательских приложений с этими демонами.
  - На обычной системе есть только одна системная шина.
  - На встраиваемых GNU/Linux системах может быть единственной шиной.
- **Сессионная**
  - Создаётся для каждого пользователя, аутентифицировавшегося в системе.
  - Для каждой такой шины запускается отдельная копия демона, посредством неё будут общаться приложения, с которыми работает пользователь.
  - Связана с X-сессией.

Инструменты

- Консольные:
  - **dbus-send** – вызов методов у сервисов D-Bus из консоли. Существуют также альтернативные утилиты для этой задачи:
    - gdbus (glib)
    - qdbus (Qt)
  - **dbus-monitor** – вывод трафика на шине D-Bus в консоль.
- Графические:
  - **D-Feet** – Показывает иерархию сервисов на шине D-Bus, позволяет вызывать у них методы.
  - **Bustle** – Записывает трафик на шине D-Bus (как и утилита “dbus-monitor”), при этом представляет информацию в виде диаграмм последовательности.

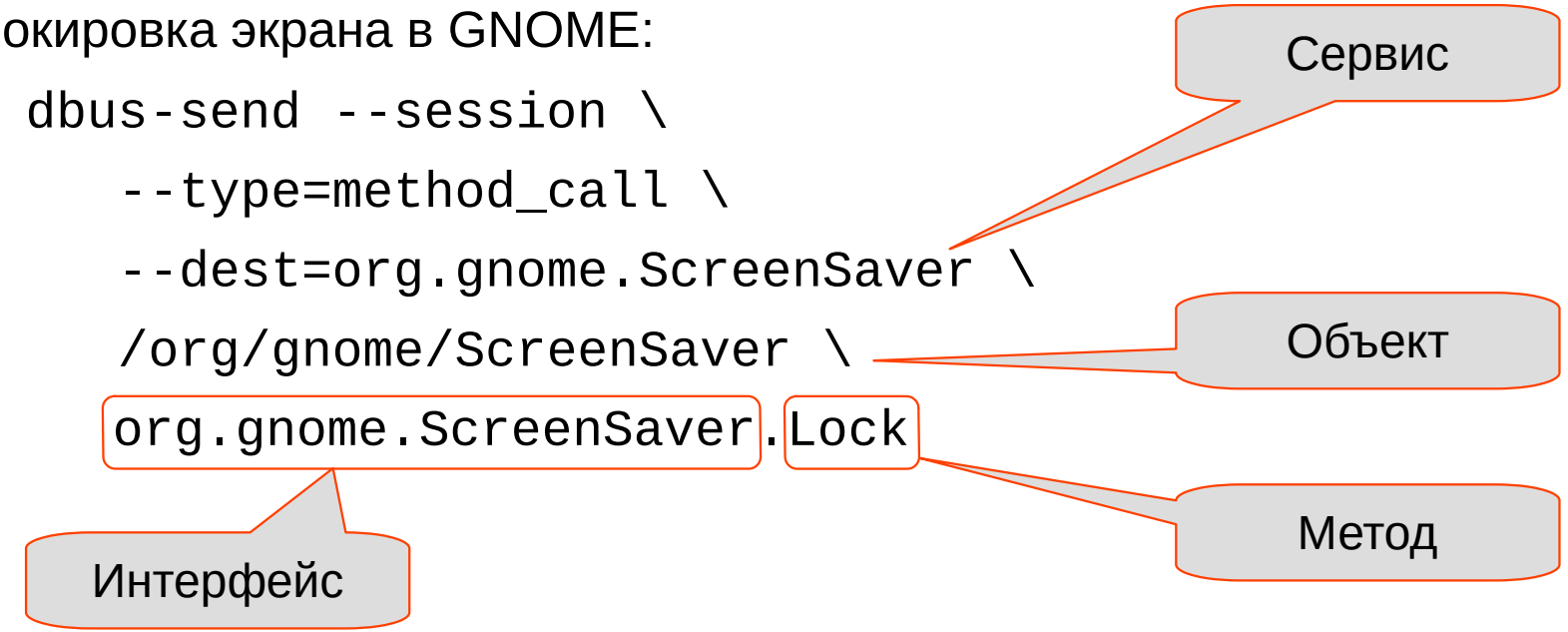
dbus-send

- Обычно не требует установки, т.к. является частью предустановленного пакета “dbus”.
- Можно выбрать системную (“--system”) или сессионную (“--session”) шину через опции.

dbus-send: Пример использования

- Блокировка экрана в GNOME:

```
dbus-send --session \
    --type=method_call \
    --dest=org.gnome.ScreenSaver \
    /org/gnome/ScreenSaver \
    org.gnome.ScreenSaver.Lock
```



- **Важно:** “dbus-send” не поддерживает все возможные типы данных D-Bus, следовательно не может вызывать некоторые методы.

dbus-send: Больше примеров

- Получение свойств объекта:

```
dbus-send --system --print-reply --dest=net.connman \
    / net.connman.Clock.GetProperties
```

- Изменение свойства:

```
dbus-send --system --print-reply --dest=net.connman \
    / net.connman.Clock.SetProperty \
    string:TimeUpdates variant:string:manual
```

- Использование стандартных интерфейсов:

```
dbus-send --system --print-reply --dest=net.connman \
    / org.freedesktop.Dbus.Introspectable.Introspect
```

```
dbus-send --system --print-reply --dest=fi.w1.wpa_supplicant1 \
    /fi/w1/wpa_supplicant1 org.freedesktop.DBus.Properties.Get \
    string:fi.w1.wpa_supplicant1 string:Interfaces
```

dbus-monitor

- Является частью предустановленного пакета “dbus”.
- Отслеживает и показывает весь трафик на шине D-Bus (включая методы и сигналы, если это предусмотрено политикой.)
- Может также фильтровать сообщения по названию интерфейса:

```
dbus-monitor --system type=signal interface=net.connman.Clock
```

dbus-monitor: Пример вывода

```
$ sudo dbus-monitor --system
signal time=1704952298.943490 sender=org.freedesktop.DBus -> destination=:1.259
serial=2 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus;
member=NameAcquired
   string ":1.259"
signal time=1704952298.943516 sender=org.freedesktop.DBus -> destination=:1.259
serial=4 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=NameLost
   string ":1.259"
method call time=1704952300.315092 sender=:1.260 -> destination=org.freedesktop.DBus
serial=1 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=Hello
method return time=1704952300.315131 sender=org.freedesktop.DBus ->
destination=:1.260 serial=1 reply_serial=1
   string ":1.260"
```

gdbus

- Обычно не требует установки, т.к. является частью библиотеки “glib”.
- Предоставляет интерфейс командной строки для D-Bus (как “dbus-send” и “dbus-monitor”.)
- Имеет больше возможностей (в том числе, поддержку сложных типов данных.)
- Интерфейс утилиты отличается от “dbus-monitor” – первым аргументом необходимо указать команду, такую, как “call” или “monitor”.
  - `gdbus call --system --dest net.connman \`
    `--object-path / --method net.connman.Clock.GetProperties`
  - `gdbus call --system --dest net.connman --object-path / \`
    `--method net.connman.Clock.SetProperty 'TimeUpdates' "<'manual'>"`
  - `dbus monitor --system --dest net.connman`
- Может отправлять сигналы:
  - `gdbus emit --session --object-path / --signal \`
    `net.connman.Clock.PropertyChanged “[‘TimeUpdates’, \<’auto’\>]”`

gdbus: Использование

- Показ уведомления на рабочем столе:

```
gdbus call --session \
           --dest org.freedesktop.Notifications \
           --object-path /org/freedesktop/Notifications \
           --method org.freedesktop.Notifications.Notify \
           my_app_name \
           42 \
           gtk-dialog-info \
           "Заголовок уведомления" \
           "Тело уведомления" \
           [] \
           {} \
           5000
```

D-Feet: Установка

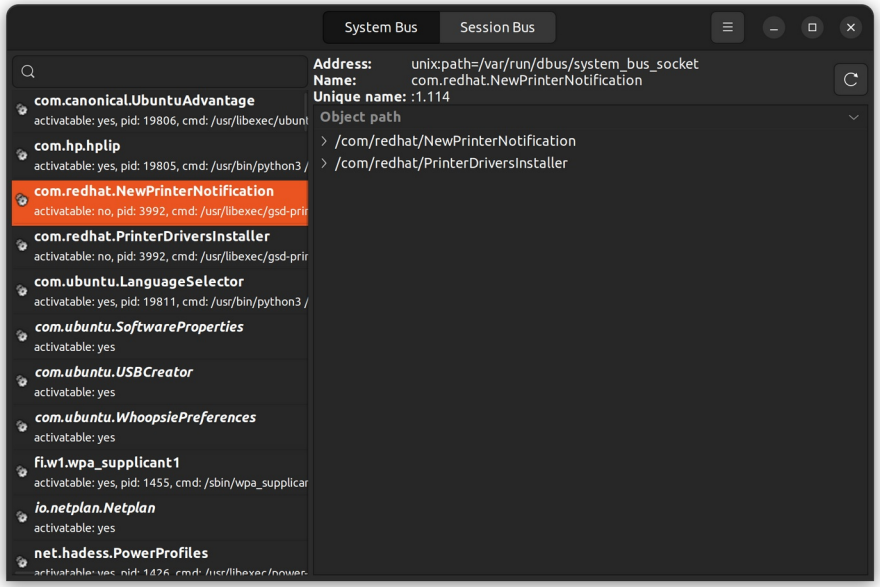
- Ubuntu:

```
sudo apt install d-feet
```

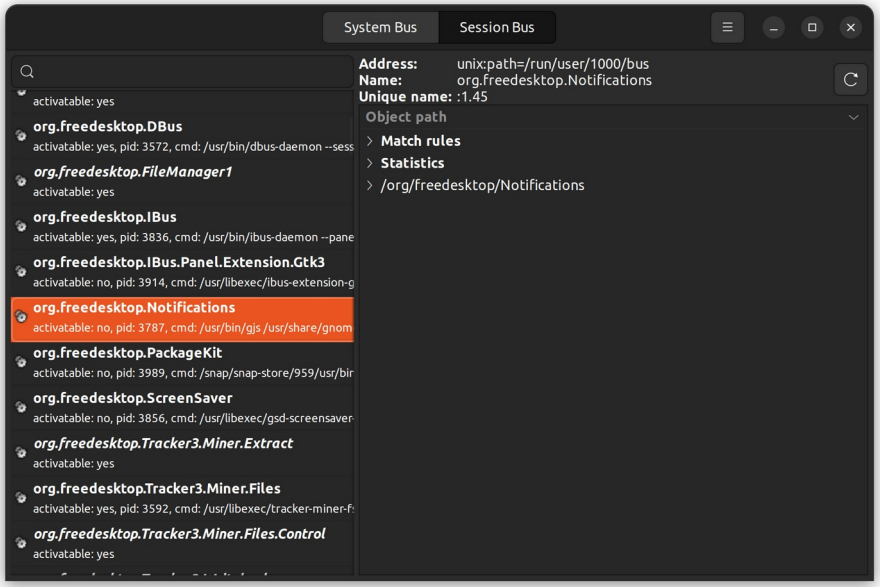
- ALT:

```
sudo apt-get install d-feet
```

D-Feet: Внешний вид (на Ubuntu)

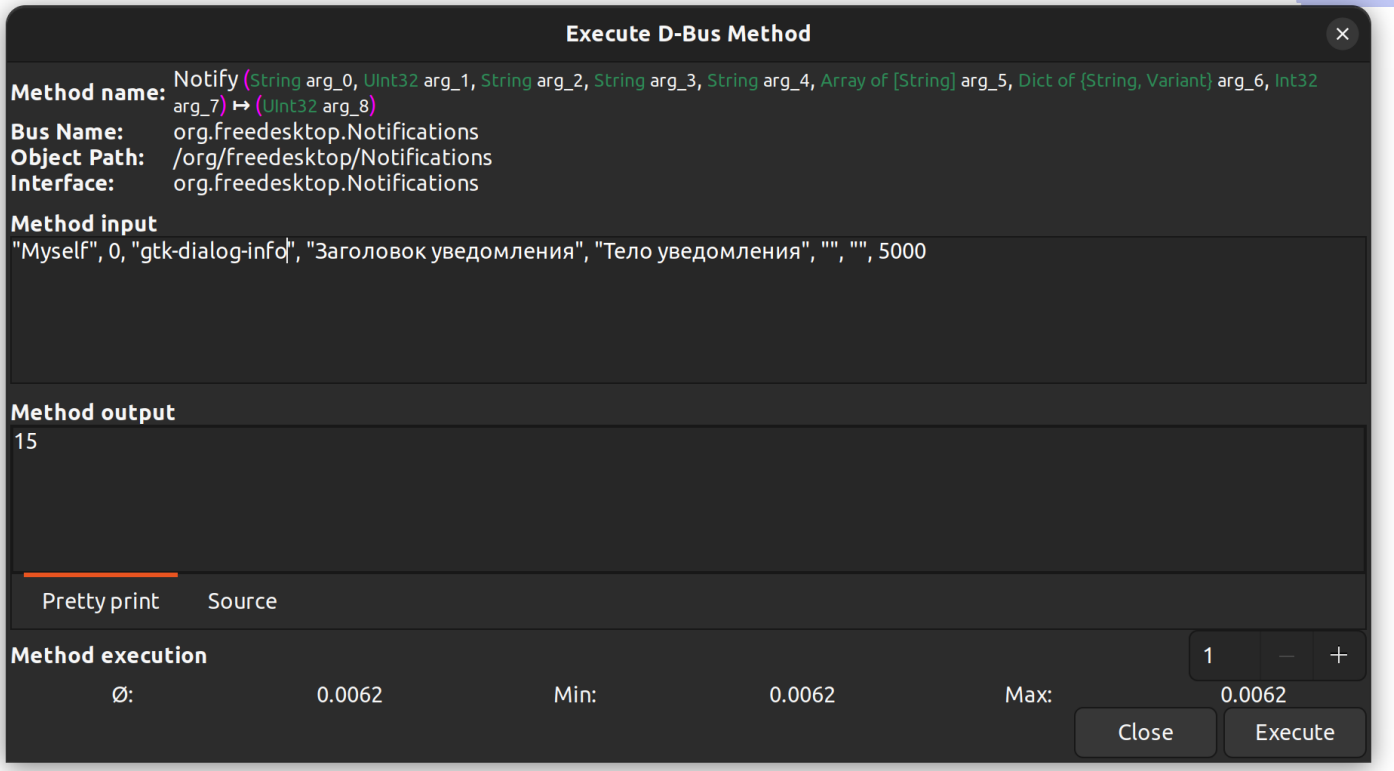


System Bus



Session Bus

D-Feet: Вызов метода



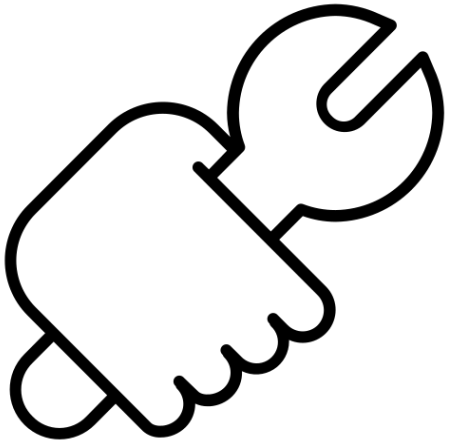
D-Feet: Параметры метода

```
"Myself", 0, "gtk-dialog-info", "Заголовок уведомления", "Тело уведомления", "", "", 5000
```

- Объяснение:
  - “Myself” – название приложения.
  - 0 – ID уведомления (0 означает автоматический номер)
  - “gtk-dialog-info” – Название иконки для уведомления.
  - “Заголовок уведомления” – эта строка показывается в заголовке.
  - “Тело уведомления” – да.
  - “” – Массив возможных действий для уведомления.
  - “” – Подсказки для уведомления.
  - 5000 – Время показа уведомления.

Проекты, использующие D-Bus

- **KDE** – окружение рабочего стола на основе библиотек Qt.
- **GNOME** – окружение рабочего стола на основе библиотек GTK.
- **SystemD** – подсистема инициализации и управления службами GNU/Linux.
- **BlueZ** – подсистема Bluetooth для GNU/Linux.
- **Pidgin** – приложение для обмена мгновенными сообщениями.
- **Network-manager** – демон для управления сетевыми интерфейсами.
- **Modem-manager** – демон для управления модемами, работает вместе с “Network-manager”.
- **Ofono** – демон, предоставляющий доступ к возможностям устройств телефонии (GSM/UMTS), таких, как модемы.
- **Connman** – тоже, что и “Network-manager”, но работает с “Ofono”.



Практика

Практика

- Платформы, инструменты и библиотеки:
  - Операционные системы: **Ubuntu 22.04** и **ALT Linux p10**
  - Язык программирования: **C**
  - Компилятор: **gcc**
  - Библиотеки: **glib**, **gio**
- Цель:
  - Разработка сервиса, умеющего обрабатывать вызовы методов по шине D-Bus.

Установка зависимостей

- Ubuntu 22.04:
  - `sudo apt install gcc libglib2.0-dev libgio3.0-cil-dev`
- ALT p10:
  - `sudo apt-get install libglib2-devel libgio-devel`

Структура проекта

- **server.c**
  - Реализация приложения, которое регистрирует на шине D-Bus сервис с необходимыми интерфейсами и обрабатывает вызовы методов.
  - При запуске уходит в бесконечный цикл ожидания и обработки запросов на шине.
  - Останавливается принудительно по Ctrl+C.
- **client.c**
  - Клиентское приложение, которое при запуске делает один запрос на созданный нами сервис, выводит на экран результат и завершается.

Функции логирования

- `g_error`
  - Критическая ошибка.
- `g_warning`
  - Предупреждение (система возможно сможет работать дальше.)
- `g_info`
  - Информационное сообщение.
- `g_debug`
  - Отладочное сообщение.

Пример использования:

```
g_warning ("server: Oh noes! No method registered for \"%s\"\n",
           method_name);
```

server.c: Подключение заголовочных файлов

```
// Библиотека Glib
#include <glib.h>

// Процедуры вывода на экран
#include <glib/gprintf.h>

// Высокоуровневые процедуры ввода/вывода,
// сетевого взаимодействия, настройки и т.п.
#include <gio/gio.h>

// Стандартная библиотека языка C.
#include <stdlib.h>
```

server.c: Описание интерфейса

```
static const gchar INTROSPECTION_XML[] =
  "<node>"
  "  <interface name='com.example.server.interface'>"
  "    <method name='getRandomJoke'>"
  "      <arg name='response' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
  "  <interface name='org.freedesktop.DBus.Properties'>"
  "    <method name='Get'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='out'/>"
  "    </method>"
  "    <method name='Set'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='in'/>"
  "    </method>"
  "    <method name='GetAll'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='values' type='a{sv}' direction='out'/>"
  "    </method>"
  "  </interface>"
  "  <interface name='org.freedesktop.DBus.Introspectable'>"
  "    <method name='Introspect'>"
  "      <arg name='xml_data' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
"</node>";
```

server.c: Описание интерфейса

```
static const gchar INTROSPECTION_XML[] =
  "<node>"
  "  <interface name='org.freedesktop.DBus.Properties'>"
  "    <method name='Get'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='out'/>"
  "    </method>"
  "    <arg name='response' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
  "  <interface name='org.freedesktop.DBus.Properties'>"
  "    <method name='Get'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='out'/>"
  "    </method>"
  "    <method name='GetAll'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='values' type='a{sv}' direction='out'/>"
  "    </method>"
  "  </interface>"
  "  <interface name='org.freedesktop.DBus.Introspectable'>"
  "    <method name='Introspect'>"
  "      <arg name='xml_data' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
"</node>";
```

Тип данных (символ)

Название константы

Квадратные скобки означают массив

server.c: Описание интерфейса

Название интерфейса

Название метода

```
static const gchar INTROSPECTIO
  "<node>"
  "  <interface name='com.example.server.interface'>"
  "    <method name='getRandomJoke'>"
  "      <arg name='response' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
  "  <interface name='org.freedesktop.DBus.Properties'>"
  "    <method name='Get'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='out'/>"
  "    </method>"
  "    <method name='Set'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='in'/>"
  "    </method>"
  "    <method name='GetAll'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='values' type='a{sv}' direction='out'/>"
  "    </method>"
  "  </interface>"
  "  <interface name='org.freedesktop.DBus.Introspectable'>"
  "    <method name='Introspect'>"
  "      <arg name='xml_data' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
"</node>";
```

server.c: Описание интерфейса

```
static const gchar INTROSPECTION_XML[] =
  "<node>"
  "  <interface name='com.exam
  "    <method name='getRandomJo
  "      <arg name='response' type='s' direction='out'/>"
  "    </method>"
  "  </interface
  "  <interface                    DBus.Pr
  "    <method name='Get'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='out'/>"
  "    </method>"
  "    <method name='Set'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      <arg name='property_name' type='s' direction='in'/>"
  "      <arg name='value' type='v' direction='in'/>"
  "    </method>"
  "    <method name='GetAll'>"
  "      <arg name='interface_name' type='s' direction='in'/>"
  "      ame='values' type='a{sv}' direction='out'/>"
  "    d>"
  "    e>"
  "  <interface name='org.freedesktop.DBus.Introspectable'>"
  "    <method name='Introspect'>"
  "      <arg name='xml_data' type='s' direction='out'/>"
  "    </method>"
  "  </interface>"
  "</node>";
```

Тип параметра

Название параметра

Направление параметра

server.c: Преобразование XML в объект

- Для хранения описания объекта D-Bus мы создадим глобальную переменную:

```
static GDBusNodeInfo *introspection_data = NULL;
```

- Значение данная переменная получит при запуске программы внутри главной функции “main”:

```
introspection_data = g_dbus_node_info_new_for_xml (INTROSPECTION_XML, NULL);
```

Наш XML, заданный выше в виде строки

server.c: Основные callback'и – 1

- `static void` **`on_bus_acquired`** `(GDBusConnection *connection, const gchar *name, gpointer user_data)`
  - Вызывается при получении доступа к шине.
- `static void` **`on_name_acquired`** `(GDBusConnection *connection, const gchar *name, gpointer user_data)`
  - Вызывается при получении имени сервиса на шине.
- `static void` **`on_name_lost`** `(GDBusConnection *connection, const gchar *name, gpointer user_data)`
  - Вызывается при потери имени на шине.

server.c: Основные callback'и – 2

- ```
  static void handle_method_call (GDBusConnection       *connection,
                                  const gchar           *sender,
                                  const gchar           *object_path,
                                  const gchar           *interface_name,
                                  const gchar           *method_name,
                                  GVariant              *parameters,
                                  GDBusMethodInvocation *invocation,
                                  gpointer               user_data)
  ```
  - Вызывается при вызове метода у объекта.
  ```

server.c: Реализация callback’ов – 1

```
static void on_name_acquired (GDBusConnection *connection,
                              const gchar     *name,
                              gpointer         user_data) {
   g_printf("name: %s\n", name);
}
```

- Данная функция просто выводит на экран имя, успешно зарегистрированное на шине.

server.c: Реализация callback'ов – 2

```
static void handle_method_call (
      GDBusConnection*       connection,
      const gchar*           sender,
      const gchar*           object_path,
      const gchar*           interface_name,
      const gchar*           method_name,
      Gvariant*              parameters,
      GDBusMethodInvocation* invocation,
      gpointer               user_data
    ) {
    // …
}
```

- Обработчик вызова метода объекта.

server.c: Реализация callback’ов – 3

- ```
  static void handle_method_call (
        // ...
      ) {
      // ...
      const gchar *uniquebusname = g_dbus_connection_get_unique_name (connection);
      // ...
  }
  ```
  - Получение уникального имени, которое было присвоено демоном D-Bus для данного подключения.
  ```

server.c: Реализация callback’ов – 4

```
static void handle_method_call (
      // ...
    ) {
    // ...
    gchar *tmp = NULL;
    g_variant_get (parameters, "(s)", &tmp);
    // ...
}
```

- Пример получения строкового параметра из запроса через функцию “g_variant_get”.

server.c: Реализация callback’ов – 5

- ```
  static void handle_method_call (
        // ...
      ) {
      // ...
      g_dbus_method_invocation_return_value (invocation, NULL);
      // ...
  }
  ```
  - Пример возврата пустого значения (“NULL”) через функцию “g_dbus_method_invocation_return_value”.
  - Вместо “NULL” здесь может быть передан указатель на объект с типом “GVariant”.
  ```

server.c: Реализация callback’ов – 6

```
static void handle_method_call (
      // ...
    ) {
    // ...
    g_dbus_method_invocation_return_error (
        invocation,
        g_quark_from_string ("server"),
        99,
        "server does not know method \"%s\"",
        method_name
    );
    // ...
}
```

- Пример возврата ошибки через функцию “g_dbus_method_invocation_return_error”.

server.c: Регистрация callback’ов

```
static const GDBusInterfaceVTable INTERFACE_VTABLE = {
  handle_method_call, // method call
  NULL,               // get_prop
  NULL                // set_prop
};
```

- Структура, хранящая в себе указатели на функции, которые должны вызываться при определённых обращениях к нашему сервису. В том числе, здесь можно видеть “handle_method_call”.
- Вызовы для управления свойствами D-Bus объекта, соответствующие “get_prop” и “set_prop”, в нашем коде не используются – заданы как “NULL”. Иными словами, объект не будет отвечать на подобные запросы.
- В данном случае “INTERFACE_VTABLE” является глобальной константой, которую мы используем далее в коде.

server.c: Главная функция “main”

- ```
  int main (int argc, char *argv[]) {
      // ...
  }
  ```
  - С функции “main” начинается выполнение программы. По завершению “main” программа также завершается.
  - Здесь мы должны создать и зарегистрировать наш D-Bus сервис.
  - Также мы должны запустить основной цикл обработки сообщений на шине D-Bus, которые адресованы нашему сервису.
  ```

server.c: Работа со старыми версиями glib

- ```
  int main(int argc, char* argv[]) {
    #if OLD_GLIB
     g_type_init();
    #endif
    // ...
  }
  ```
  - Если используется старая версия Glib, то мы должны вызывать функцию “g_type_init” в начале “main”.
  ```

main.c: Главный цикл сервиса

- ```
  int main(int argc, char* argv[]) {
    // …
    GMainLoop *loop = g_main_loop_new (NULL, FALSE);
    // ...
  }
  ```
  - Переменная “loop” хранит в себе объект, описывающий “главный цикл” нашего сервиса – в этом цикле сервис будет слушать и обрабатывать сообщения, приходящие по шине D-Bus.
  ```

server.c: Регистрация имени

```
guint owner_id = g_bus_own_name (
                                  G_BUS_TYPE_SESSION,
                                  "com.example.server",
                                  G_BUS_NAME_OWNER_FLAGS_NONE,
                                  on_bus_acquired,
                                  on_name_acquired,
                                  on_name_lost,
                                  NULL,
                                  (GDestroyNotify) NULL );
```

- Регистрация сервиса в D-Bus на сессионной шине с указанным именем “com.example.server”. Функция возвращает идентификатор, по которому можно отменить регистрацию имени.

server.c: Запуск основного цикла

- `g_main_loop_run (loop);`
  - Функция запускает основной цикл сервиса D-Bus.
  - Выйти из цикла можно, вызвав функцию `g_main_loop_quit` изнутри основного цикла (например, в обработчике запроса D-Bus.)

server.c: Освобождение ресурсов

- `g_bus_unown_name (owner_id);`
  `g_dbus_node_info_unref (introspection_data);`
  - По завершению цикла D-Bus, необходимо вернуть ресурсы системе. Это делается через отмену регистрации имени (`g_bus_unown_name`) и удаление данных о сервисе (`g_dbus_node_info_unref`).

client.c: Подключение библиотек

```
// Библиотека Glib
#include <glib.h>

// Процедуры вывода на экран
#include <glib/gprintf.h>

// Высокоуровневые процедуры ввода/вывода,
// сетевого взаимодействия, настройки и т.п.
#include <gio/gio.h>

// Стандартная библиотека языка C.
#include <stdlib.h>
```

client.c: Работа со старыми версиями glib

- ```
  int main(int argc, char* argv[]) {
      #if OLD_GLIB
      g_type_init();
      #endif
      // ...
  }
  ```
  - Если используется старая версия Glib, то мы должны вызывать функцию “g_type_init” в начале “main”.
  ```

client.c: Открытие подключения к шине

```
int main(int argc, char* argv[]) {
    // ...
    GError *error = NULL;
    GDBusConnection *con = g_bus_get_sync (G_BUS_TYPE_SESSION, NULL, &error);
}
```

- Открытие подключения к сессионной шине D-Bus. Если при подключении возникает ошибка, то она будет записана в структуру `GError` (переменная “error”).

client.c: Формирование сообщения

```
GDBusMessage *msg = g_dbus_message_new_method_call (
    "com.example.server",
    "/com/example/server/obj",
    "com.example.server.interface",
    "getRandomJoke");
```

- Функция “`g_dbus_message_new_method_call`” создаёт новое сообщение, которое можно отправить на сервис.

client.c: Запрос

```
int main(int argc, char* argv[]) {
    // ...
    GDBusMessage *resp = g_dbus_connection_send_message_with_reply_sync (
        con,
        msg,
        G_DBUS_SEND_MESSAGE_FLAGS_NONE,
        -1,
        NULL,
        NULL,
        &error);
    );
}
```

- Пример синхоронного запрос на шину D-Bus. Если возникает ошибка, она сохраняется в структуру “error” (GError).

client.c: Получение и обработка ответа

- ```
  gchar* value;
  g_variant_get(g_dbus_message_get_body(resp),
                "(s)",
                &value);
  printf("%s\n", value);
  ```
  - Из ответа сервиса мы получаем строку и выводим её на экран.
  ```

client.c: Освобождение ресурсов

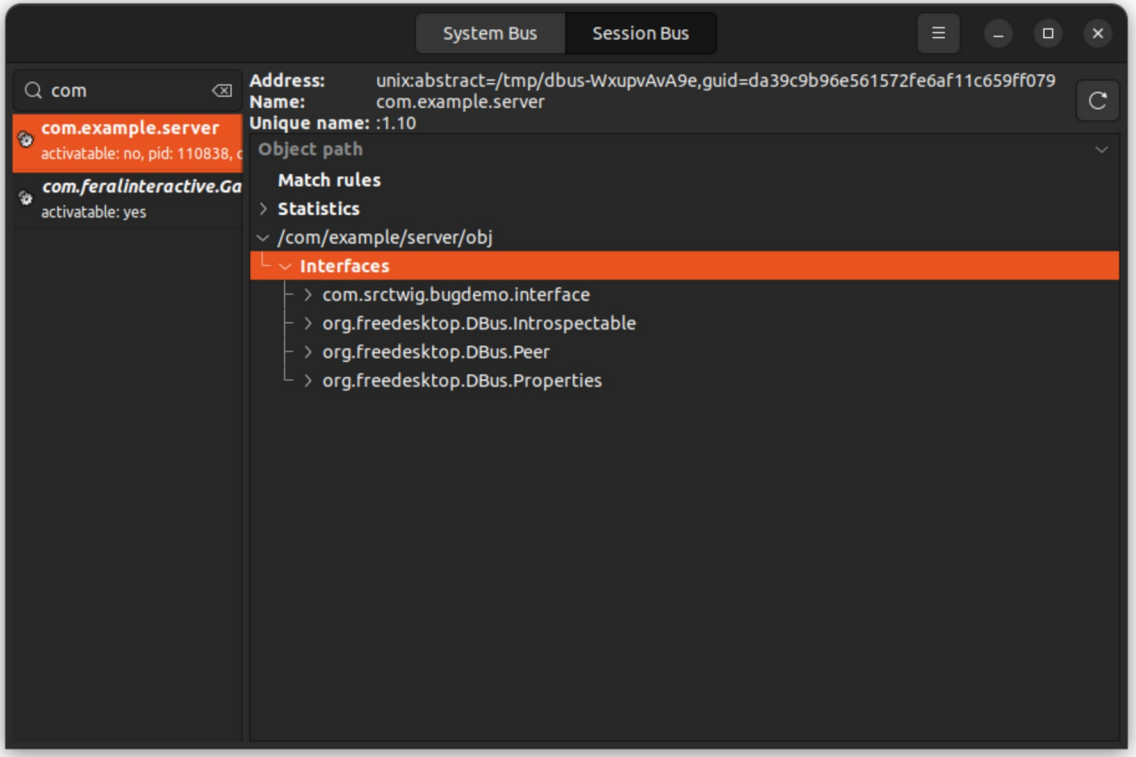
- ```
  int main(int argc, char* argv[]) {
      // ...
      g_object_unref (con);
      return EXIT_SUCCESS;
  }
  ```
  - Подключение к D-Bus закрывается, программа завершается.
  ```

Сборка проекта

- Получение информации о необходимых опциях компилятора:
  - **CFLAGS**=$(pkg-config --cflags glib-2.0)
  - **LIBS**=$(pkg-config --libs glib-2.0 gio-2.0)
- Запуск компилятора:
  - gcc -o server $**CFLAGS** server.c $**LIBS**
  - gcc -o client $**CFLAGS** client.c $**LIBS**
- В примере кода приложен Makefile, который позволяет удобно собрать клиент и сервер:
  - make

Тестирование сервера

- Запуск:
  - `sudo ./server`



Источники

- Mylène Josserand, **“Understanding D-Bus”**: https://bootlin.com/pub/conferences/2016/meetup/dbus/josserand-dbus-meetup.pdf

Спасибо за внимание!

- Артём “avp” Попцов
  - Персональная страница: https://memory-heap.org
  - Email: poptsov.artyom@gmail.com
  - Mastodon: https://fosstodon.org/@avp
- Лицензия на презентацию:
  - CC-BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0/