Роман Химов

Посредник D-BUS

Понятие настольной среды подразумевает нечто цельное и тесно переплетенное между собой, однако среда формируется приложениями, а они как-то должны друг с другом общаться, интегрируясь в нее. В мире KDE для этого давно и небезуспешно используется DCOP, но вот в других настольных средах (например, GNOME) дела все время обстояли не так хорошо.

Конечно, существовала возможность коммуникации посредством CORBA, SOAP или XML-RPC, но первый очень уж тяжеловат и неудобен (причем, надо заметить, что и KDE, и GNOME прошли этап его использования за время своего существования), а оставшиеся не так быстры, как хотелось бы, да и предназначены они для совсем других целей.

До недавнего времени GNOME использовал Bonobo, который, в свою очередь, был основан на CORBA, но в связи с зависимостью от GObject более Bonobo нигде не использовался, да и тяжесть CORBA по-прежнему давила на всю среду. А ведь хотелось бы, чтобы свободно общаться между собой могли не только приложения одной среды, но и разных.

На это и направлен новый стандарт D-BUS, разрабатываемый обществом freedesktop.org. Продукт легковесен и очень быстр в работе, он не зависит от конкретной среды, но при этом прекрасно интегрируется в каждую из них. D-BUS также легко доступен из разных программных сред (Glib, Java (GCJ), Mono, Qt, Python) и прозрачен для сети.

Что такое D-BUS?

D-BUS — это механизм IPC (InterProcess Communication), который предоставляет шину для передачи сообщений, ну а если быть точнее, то сразу несколько шин. Первая и самая главная — системная шина, она создается уже при старте демона D-BUS, и с ее помощью происходит общение различных демонов. Она хорошо защищена от посторонних, и пользовательские приложения, хоть и могут подключаться к ней, все же будут значительно ограничены в том, какие сообщения они смогут туда посылать (в то же время они могут многое «услышать»).

Реальная же рабочая лошадка D-BUS — сессионная шина, создаваемая для любого пользователя, авторизующегося в системе. Для каждой такой шины запускается отдельная копия демона, и именно посредством нее будут общаться приложения, с которыми работает этот пользователь.

Каждое сообщение D-BUS, передаваемое по шине, имеет своего отправителя и своего получателя, их адреса называются путями объектов, поскольку D-BUS предполагает, что каждое приложение состоит из набора объектов, а сообщения пересылаются не между приложениями, а между объектами этих самых приложений. Для идентификации объектов используются пути, именуемые в стиле Unix. Так, например, сам D-BUS доступен по адресу /org/freedesktop/DBus.

Каждый объект может поддерживать один или более интерфейсов, которые представлены здесь в виде именованных групп методов и сигналов — аналогично интерфейсам Glib, Qt или Java. Именуются они привычным для ОО-программистов образом, например рассматриваемый D-BUS экспортирует интерфейс org.freedesktop.DBus.

Однако объектов и интерфейсов маловато для реализации некоторых интересных возможностей, и D-BUS также предусматривает концепцию сервисов. Сервис — уникальное местоположение приложения на шине. При запуске приложение регистрирует один или несколько сервисов, которыми оно будет владеть до тех пор, пока самостоятельно не освободит, до этого момента никакое другое приложение, претендующее на тот же сервис, занять его не сможет. Именуются сервисы аналогично интерфейсам, а сам D-BUS экспортирует, соответственно, сервис org.freedesktop.DBus.

Принципы работы

Сервисы делают доступной еще одну функцию — запуск необходимых приложений в случае поступления сообщений для

них. Для этого должна быть включена автоактивация, и в конфигурации D-BUS за этим сервисом должно быть закреплено одно приложение. Тогда D-BUS сможет его запустить при появлении сообщения.

После закрытия приложения ассоциированные сервисы также разрегистрируются, а D-BUS посылает сигнал о том, что сервис закрыт. Другие приложения могут получать такие сигналы и соответствующим образом реагировать.

После подключения к шине приложение должно указать, какие сообщения оно желает получать, путем добавления масок совпадений (matchers). Маски представляют собой наборы правил для сообщений, которые будут доставляться приложению, фильтрация может основываться на интерфейсах, путях объектов и методах. Таким образом, приложения будут получать только то, что им необходимо, проблемы доставки в этом случае берет на себя D-BUS.

Сообщения в D-BUS бывают четырех видов: вызовы методов, результаты вызовов методов, сигналы и ошибки. Первые предназначены для выполнения методов над объектами, подключенными к D-BUS; посылая такое сообщение, вы выдаете задание объекту, а он после обработки обязан возвратить вам либо результат вызова, либо ошибку через сообщения соответствующих типов. Сигнальные же сообщения, как им и полагается, ничуть не заботятся о том, что и как делается объектами, они вольны воспринимать их как угодно (равно как и не получать их вовсе).

Естественно, для нормальной работы вызовов методов между различными средами/языками D-BUS стандартизирует типы передаваемых данных (параметров). Мы не будем их перечислять, поскольку это не имеет смысла, однако отметим, что интерфейсы D-BUS строго типизированы, и D-BUS всегда выполняет проверку корректности передаваемых по интерфейсу данных, не доверяя никому.

Системная шина D-BUS защищена от вредных приложений. Каким же образом это реализовано? Ответ располагается по адресу /etc/dbus-1/system.d/ — там можно обнаружить различные конфигурационные файлы, с помощью которых осуществляется очень гибкая настройка доступа к тем или иным сервисам и интерфейсам. Впрочем, если этой гибкости мало, можно сделать еще больше, подключив возможность использования контекстов безопасности SELinux. В базовый протокол D-BUS также встроены методы аутентификации клиентов, обращающихся с вызовами, однако они практически не используются.

Пример работы

Посмотрите на следующий пример использования механизма D-BUS из Python: код взят из документации к другому любимому нами компоненту — HAL:

```
# !/usr/bin/python
import gtk
import dbus
def device_added(interface, signal_name, service, path, message):
[udi] = message.get_args_list ()
print 'Device %s was added'%udi
def device_removed(interface, signal_name, service, path, message):
[udi] = message.get_args_list ()
print 'Device %s was removed'%udi
bus = dbus.Bus (dbus.Bus.TYPE_SYSTEM)
hal_service = bus.get_service ('org.freedesktop.Hal')
hal_manager = hal_service.get_object
('/org/freedesktop/Hal/Manager',
'org.freedesktop.Hal.Manager')
bus.add_signal_receiver (device_added,
'DeviceAdded',
'org.freedesktop.Hal.Manager',
'org.freedesktop.Hal',
'/org/freedesktop/Hal/Manager')
bus.add_signal_receiver (device_removed,
'DeviceRemoved',
'org.freedesktop.Hal.Manager',
'org.freedesktop.Hal',
'/org/freedesktop/Hal/Manager')
gtk.main()
```

Запустите такой скрипт и поиграйте с устройствами — подключите/отключите мышку, принтер, сканер, еще что-нибудь. Все эти действия будут сопровождаться сообщениями от скрипта, поскольку он ловит сигналы device_added и device_removed от HAL через D-BUS (на системной шине). Согласитесь, отловить эти сообщения совсем несложно, зато представьте, сколько всего полезного можно сделать (если чуть-чуть расширить скрипт на предмет более подробных расспросов HAL об устройстве)!

Применения

Разработчики KDE пока еще только начали осваивать D-BUS, что, в общем-то, и понятно: DCOP не намного хуже D-BUS (надо отметить, что последний учел опыт первого в разработке), но с ним KDE работает уже давно. Однако есть проект перевода DCOP на использование D-BUS в качестве транспорта, да и, как уже было сказано, освоение уже пошло (все, что относится к взаимодействию с HAL в KDE, также протекает посредством D-BUS).

А вот GNOME уже активно использует D-BUS в своих приложениях, одни примеры его применения вместе с HAL многого стоят (что интересно, HAL без D-BUS был бы довольно скучен, но и D-BUS смог по-настоящему показать свои прелести только на HAL). Однако этим его использование отнюдь не ограничивается — во-первых, GNOME сейчас активно избавляется от Bonobo, и здесь D-BUS начинает тихо и без лишнего шума выполнять свою основную работу, пусть и не с первого взгляда заметную пользователю.

Во-вторых, наличие удобного транспорта уже навело некоторых разработчиков на мысли о том, что с его помощью можно легко обобщать многие части общего кода приложений, создавая сервисы D-BUS, которые фактически становятся очень интересной альтернативой обычным разделяемым библиотекам, которые, в свою очередь, будут доступны из любых языков и сред. Поэтому, как и в случае с HAL, все самое интересное еще впереди.