Data sharing

Contents

This page describes design patterns that can be used for inter-process communication, particularly between applications and agents in the same or different app-bundles. We consider a situation in which one or more ***consumers*** receive information from one or more ***providers***; we refer to the consumer and provider together as ***peers***.

Use cases

- Points of interest[1] should use one of these patterns
- Sharing[2] could use one of these patterns
- Global search (see ConceptDesigns[3]) currently carries out the equivalent of interface discovery[4] by reading the manifest directly, but other than that it is similar to Query-based access via D-Bus

Selecting an initiator

The first design question is which peer should initiate the connection (the ***initiator***) and which one should not (the ***responder***).

When the connection is first established, the initiator must already be running. However, the responder does not necessarily need to be running: in some cases it could be started automatically.

Some guidelines:

- If one of the peers is a HMI (user interface) that only appears when it is started by the user, but the other is an agent, then the HMI should be the initiator and the agent should be the responder.

---

[1] https://martyn.pages.apertis.org/apertis-website/concepts/points_of_interest/
[2] https://martyn.pages.apertis.org/apertis-website/concepts/sharing/
[3] https://martyn.pages.apertis.org/apertis-website/concepts/global-search/
[4] https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/

- If one of the peers is assumed to be running already, but the other can be auto-started on-demand, then the peer that is running already should be the initiator, and the peer that can be auto-started should be the responder.
- If the connection is normally only established when one of the peers receives user input, then that peer should be the initiator.
- If there is no other reason to prefer one direction over the other, the consumer is usually the initiator.

Where there are multiple consumers or multiple providers, base the decisions on which of these things is expected to be most frequent among consumers and among providers.

Discovery

Each initiator carries out Interface discovery[5] to find implementations of the responder. If the initiator is the consumer, the interface that is discovered might have a name like `com.example.PointsOfInterestProvider`. If the initiator is the provider, the interface that is discovered might have a name like `com.example.DebugLogConsumer`.

If the responder is known to be a platform service, then interface discovery is unnecessary and should not be used. Instead, the initiator(s) may assume that the responder exists. Its API documentation should include its well-known bus name, and the object paths and interfaces of its "entry point" object.

Connection

Each initiator initiates communication with each responder by sending a D-Bus method call.

We recommend that each responder has a D-Bus well-known name matching its app ID, using the reversed-DNS-name convention described in the Applications design document. For example, if Collabora implemented a `PointsOfInterestProvider` that advertised the locations of open source conferences, it might be named `uk.co.collabora.ConferenceList`. The responder should be "D-Bus activatable": that is, it should install the necessary D-Bus and systemd files so that it can be started automatically in response to a D-Bus message. To make this straightforward, we recommend that the platform or the app-store should generate these automatically from the application manifest.

Each interface may define its own convention for locating D-Bus objects within an implementation, but we recommend the conventions described in the freedesktop.org Desktop Entry specification[6], summarized here:

---

[5] https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/

[6] http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#interfaces

- the responder exports a D-Bus object path derived from its app ID (well-known name) in the obvious way, for example `uk.co.collabora.ConferenceList` would have an object at `/uk/co/collabora/ConferenceList`
- the object at that object path implements a D-Bus interface with the same name that was used for interface discovery, for example `com.example.PointsOfInterestProvider`
- the object at that object path may implement any other interfaces, such as `org.freedesktop.Application` and/or `org.freedesktop.DBus.Properties`

If the responder is a platform component, then it does not have an app ID, but it should have a documented well-known name following the same naming convention. If it is a platform component standardized by Apertis, its name should normally be in the `org.apertis.*` namespace. If it implements a standard interface defined by a third party and that interface specifies a well-known name to be used by all implementations (such as `org.freedesktop.Notifications`), it should use that standardized well-known name. If it is a vendor-specific component, its name should be in the vendor's namespace, for example `com.bosch.*`.

Communication

There are several patterns which could be used for the actual communication.

If the communication is expected to be relatively infrequent (an average of several seconds per message, rather than several messages per second) and convey reasonably small volumes of data (bytes or kilobytes per message, but not megabytes), and the latency of D-Bus is acceptable, we recommend that the initiator and responder use D-Bus to communicate.

If the communication is frequent or high-throughput, or low latency is required, we recommend the use of an out-of-band stream.

Publish/subscribe via D-Bus

This pattern is very commonly used when the initiator is the consumer, the message and data rates are suitable for D-Bus, and the communication continues over time.

- The consumer can receive the initial state of the provider by calling a method such as ListPointsOfInterest(), or by retrieving its D-Bus properties using GetAll(). This method call is often referred to as *state recovery*.
- The provider can notify all consumers of changes to its state by emitting broadcast signals, or notify a single consumer by using unicast signals. The consumer is expected to connect D-Bus signal handlers *before* it calls the initial method, to avoid missing events.
- We recommend that the provider should hold its state on disk or in memory so that it can provide state recovery. However, if there is a strong reason for a particular interaction to use a "carousel[7]" model in which

---

[7] https://en.wikipedia.org/wiki/Data_and_object_carousel

state is not available, this can be modelled by having the initial method call activate the provider, but not return any state.

- For efficiency, the design of the provider should ensure that the consumer can operate correctly by connecting to signals, then making the state recovery method call once. For robustness, the design of the provider should ensure that calling the state recovery method call at any time would give a correct result, consistent with the state changes implied by signals.
- If required, the consumer can control the provider by calling additional D-Bus methods defined by the interface (for example an interface might define Pause(), Resume() and/or Refresh() methods)

A complete interface for the provider might look like this (pseudocode):

```
interface com.example.ThingProvider:    /* (xy) represents whatever data structure is needed */        method ListThings() -> a(xy): things        signal ThingAdded(x: first_attribute, y: second_attribute)      signal ThingRemoved(x: first_attribute, y: second_attribute)     method Refresh() -> nothing
```

Query-based access via D-Bus

This pattern is commonly used where the initiator is the consumer and the interface is used for a series of short-lived HTTP-like request/response transactions, instead of an ongoing stream of events or a periodically updated state.

- The consumer sends a request to the provider via a D-Bus method call. This is analogous to a HTTP GET or POST operation, and can contain data from the consumer.
- The provider sends back a response via the D-Bus method response.

For example, a simple search interface might look like this (pseudocode):

```
interface com.example.SearchProvider:    /* Return a list of up to @max_results file:/// URIs with names containing @name_contains, each no larger than @max_size bytes */ method FindFilesMatching(s: name_contains, t: max_size, u: max_results) -> as: file_uris
```

(This is merely a simple example; a more elaborate search interface might consider factors like paging through results.)

Provider-initiated push via D-Bus

If the initiator is the provider and the data/message rates are suitable for D-Bus, the consumer could implement an interface that receives “pushed” events from the provider:

- the provider can send data by calling a method such as AddPointsOfInterest()

- if required, the consumer can influence the provider(s) by emitting broadcast or unicast D-Bus signals defined by the interface (for example an interface might define PauseRequested, ResumeRequested and/or RefreshRequested signals)

A complete interface for the consumer might look like this (pseudocode):

```
interface com.example.ThingReceiver:    /* (xy) represents whatever data struc-
ture is needed */     method AddThings(a(xy): things) -> nothing     signal Re-
freshRequested()
```

This pattern is unusual, and reversing the initiator/responder roles should be considered.

Consumer-initiated pull via a stream

If the initiator is the consumer and the data/message rates make D-Bus unsuitable, the provider could implement an interface that sends events into an out-of-band stream that is provided by the consumer when it initiates communication, using the D-Bus type “h” (file-handle) for file descriptor passing. For instance, in GDBus, the “_with_unix_fd_list” versions of D-Bus APIs, such as g_dbus_connection_call_with_unix_fd_list(), work with file descriptor passing.

- The consumer should create a pipe (for example using pipe2()), keep the read end, and send the write end to the provider.
- If required, the provider may send additional information, such as a filter to receive only a subset of the available records.
- The consumer may pause receiving data by not reading from the pipe. The provider should add the pipe to its main loop in non-blocking mode; it will receive write error EAGAIN if the pipe is full (paused). The provider must be careful to write a whole record at a time: even if it received EAGAIN part way through a record and skipped subsequent records, it must finish writing the partial record before doing anything else. Otherwise, the structure of the stream is likely to be corrupted.
- If there are n providers, the consumer would read from n pipes, and could receive new records from any of them.
- If there are m consumers, the provider would have m pipes, and would normally write each new record into each of them.
- The consumer may stop receiving data by closing the pipe. The provider will receive write error EPIPE, and should respond by also closing that pipe.
- If required, the consumer could control the provider by calling additional methods. For instance, the interface might define a ChangeFilter() method.

The advantages of this design are its high efficiency and low latency. The major disadvantage of this design is that the provider and consumer need to agree on a framing and serialization protocol with which they can write records into the stream and read them out again. Designing the framing and serialization protocol is part of the design of the interface.

For the serialization protocol, they might use binary TPEG records, a fixed-length packed binary structure, a serialized GVariant of a known type such as G_VARIANT_TYPE_VARIANT, or even an XML document. If streams in the same format might cross between virtual machines or be transferred across a network, interface designers should be careful to avoid implementation-dependent encodings such as numbers with unknown endianness, types with unknown byte size, or structures with implementation-dependent padding. If there is no well-established encoding, we suggest GVariant as a reasonable option.

For the framing protocol, the serialization protocol might provide its own framing (for example, fixed-length structures of a known length do not need framing), or the interface might document the use of an existing framing protocol such as netstrings[8], or its own framing/packetization protocol such as "4-byte little-endian length followed by that much data".

Interface designers should also note that there is no ordering guarantee between different pipes or sockets, and in particular no ordering guarantee between the D-Bus socket and the out-of-band pipe: if a provider sends messages on two different pipes, there they will not necessarily be received in the same order they were sent.

A complete interface might look like this (pseudocode):

```
interface  com.example.RapidThingProvider:          /*  Start  receiving  bi-
nary Thing objects and write them into      * @file_descriptor, until writ-
ing fails.       *       * The provider should ignore SIGPIPE, and write to
* @file_descriptor in non-blocking mode. If a write fails with       * EA-
GAIN, the provider should pause receiving records until    * the pipe is ready for read-
ing again. If a write fails with    * EPIPE, this indicates that the pipe has been closed, and
* the provider must stop writing to it.      *      * Arguments:      * @fil-
ter: the things to receive    * @file_descriptor: the write end of a pipe, as pro-
duced           *          by  pipe2()           */         method  Provide-
Things((some   data   structure):   filter,   h:   file_descriptor)   ->   nothing
method ChangeFilter((some data structure): new_filter) -> nothing
```

Provider-initiated push via a stream

If the initiator is the provider and the data/message rates make D-Bus unsuitable, the consumer could implement an interface that receives events from an out-of-band stream that is provided by the provider when it initiates communication, again using the D-Bus type "h" (file-handle) for file descriptor passing.

---

[8] https://en.wikipedia.org/wiki/Netstring

- The provider should create a pipe (for example using pipe2()), keep the write end, and send the read end to the provider.
- The consumer may pause receiving data by not reading from the pipe. The provider should add the pipe to its main loop in non-blocking mode; it will receive write error EAGAIN if the pipe is full (paused). The provider must be careful to write a whole record at a time, even if it received EAGAIN part way through a record and skipped subsequent records.
- If there are n providers, the consumer would read from n pipes, and could receive new records from any of them.
- If there are m consumers, the provider would have m pipes, and would normally write each new record into each of them.
- The consumer may stop receiving data by closing the pipe. The provider will receive write error EPIPE, and should respond by also closing that pipe.

As with its "pull" counterpart, the major disadvantage of this design is that the provider and consumer need to agree on a framing and serialization protocol. In addition, there is once again no ordering guarantee between different pipes or sockets.

A complete interface might look like this (pseudocode):

```
interface com.example.RapidThingReceiver:    /* @file_descriptor is the read end of a pipe */
method ReceiveThings(h: file_descriptor) -> nothing
```

Bidirectional communication via D-Bus

If required, the consumer could provide feedback to the provider by adding additional D-Bus methods and signals to the interface. For example, the Change-Filter method described above can be viewed as feedback from the consumer to the provider.

To avoid dependency loops and the potential for deadlocks, we recommend a design where method calls always go from the initiator to the responder, and method replies and signals always go from the responder back to the initiator.

Bidirectional communication via a socket or pair of pipes

If required, the consumer could provide high-bandwidth, low-latency feedback to the provider by using file descriptor passing to transfer either an AF_UNIX socket or a pair of pipes (the read end of one pipe, and the write end of another), and using the resulting bidirectional channel for communication.

We recommend that this is avoided where possible, since it requires the interface to specify a bidirectional protocol to use across the channel, and designing bidirectional protocols that will not deadlock is not a trivial task. Peer-to-peer D-Bus is one possibility for the bidirectional protocol.

As with unidirectional pipes, there is no ordering guarantee between different pipes or sockets.

Resuming communication

If the system is restarted and the previously running applications are restored, and the interface is one where resuming communication makes sense, we recommend that the original initiator re-initiates communication. This would normally be done by repeating interface discovery[9].

In a few situations it might be preferable for the original initiator to store a list of the responders with which it was previously communicating, so that it can resume communications with exactly those responders.

Stored state

In some interfaces, the provider has a particular state stored in-memory or on-disk at any given time, and the inter-process communication works by providing enough information that the consumer can reproduce that state. This approach is recommended, particularly for publish/subscribe interfaces, where it is conventionally what is done.

If implementations of a publish/subscribe interface are not required to offer full state-recovery, the interface's documentation should specifically say so. The normal assumption should be that state-recovery exists and works.

In the interfaces other than the publish/subscribe model, the initial state may be replayed at the beginning of communication by assuming that the consumer has an empty state, and sending the same data that would normally represent addition of an item or event, either as-is or with some indication that this event is being "replayed". For example, in Consumer-initiated pull via a stream, the provider would queue all currently-known items for writing to the stream as soon as the connection is opened. The interface's documentation should specify whether this is done or not.

In interfaces where the provider is stateless and has "carousel[10]" behaviour, the consumer may cache past items/events in memory or on disk for as long as they are considered valid.

Similarly, if a provider that receives items from a carousel implements an interface that expects it to store state, the provider may cache past items/events in memory or on disk for as long as they are considered valid, so that they can be provided to the consumer.

---

[9] https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/

[10] https://en.wikipedia.org/wiki/Data_and_object_carousel