



Data sharing

Contents

2	Use cases	2
3	Selecting an initiator	2
4	Discovery	3
5	Connection	3
6	Communication	4
7	Publish/subscribe via D-Bus	4
8	Query-based access via D-Bus	5
9	Provider-initiated push via D-Bus	6
10	Consumer-initiated pull via a stream	6
11	Provider-initiated push via a stream	8
12	Bidirectional communication via D-Bus	8
13	Bidirectional communication via a socket or pair of pipes	8
14	Resuming communication	9
15	Stored state	9

This page describes design patterns that can be used for inter-process communication, particularly between applications and background services 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:

¹https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/points_of_interest/

²https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/sharing/

³<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/global-search/>

⁴https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_framework/interface_discovery/

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

⁵https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_framework/interface_discovery/

71 Each interface may define its own convention for locating D-Bus objects
72 within an implementation, but we recommend [the conventions described in the](http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#interfaces)
73 [freedesktop.org Desktop Entry specification](http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#interfaces)⁶, summarized here:

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

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

90 Communication

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

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

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

99 Publish/subscribe via D-Bus

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

- 103 • The consumer can receive the initial state of the provider by calling a
104 method such as `ListPointsOfInterest()`, or by retrieving its D-Bus proper-
105 ties using `GetAll()`. This method call is often referred to as *state recovery*.
- 106 • The provider can notify all consumers of changes to its state by emitting
107 broadcast signals, or notify a single consumer by using unicast signals.

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

108 The consumer is expected to connect D-Bus signal handlers *before* it calls
109 the initial method, to avoid missing events.

- 110 • We recommend that the provider should hold its state on disk or in mem-
111 ory so that it can provide state recovery. However, if there is a strong
112 reason for a particular interaction to use a “[carousel](#)⁷” model in which state
113 is not available, this can be modelled by having the initial method call ac-
114 tivate the provider, but not return any state.
- 115 • For efficiency, the design of the provider should ensure that the consumer
116 can operate correctly by connecting to signals, then making the state
117 recovery method call once. For robustness, the design of the provider
118 should ensure that calling the state recovery method call at any time
119 would give a correct result, consistent with the state changes implied by
120 signals.
- 121 • If required, the consumer can control the provider by calling additional
122 D-Bus methods defined by the interface (for example an interface might
123 define Pause(), Resume() and/or Refresh() methods)

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

```
125 interface com.example.ThingProvider:    /* (xy) represents whatever data struc-  
126 ture is needed */                      method ListThings() -> a(xy): things      sig-  
127 nal ThingAdded(x: first_attribute, y: second_attribute)      signal ThingRe-  
128 moved(x: first_attribute, y: second_attribute)      method Refresh() -> nothing
```

129 Query-based access via D-Bus

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

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

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

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

142 (This is merely a simple example; a more elaborate search interface might con-
143 sider factors like paging through results.)

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

144 **Provider-initiated push via D-Bus**

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

- 148 • the provider can send data by calling a method such as `AddPointsOfIn-`
149 `terest()`
- 150 • if required, the consumer can influence the provider(s) by emitting broad-
151 `cast` or `unicast` D-Bus signals defined by the interface (for example an inter-
152 `face` might define `PauseRequested`, `ResumeRequested` and/or `RefreshRe-`
153 `quested` signals)

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

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

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

160 **Consumer-initiated pull via a stream**

161 If the initiator is the consumer and the data/message rates make D-Bus un-
162 suitable, the provider could implement an interface that sends events into an
163 out-of-band stream that is provided by the consumer when it initiates commu-
164 nication, using the D-Bus type “h”(file-handle) for file descriptor passing. For
165 instance, in GDBus, the “`_with_unix_fd_list`” versions of D-Bus APIs, such as
166 `g_dbus_connection_call_with_unix_fd_list()`, work with file descriptor pass-
167 ing.

- 168 • The consumer should create a pipe (for example using `pipe2()`), keep the
169 read end, and send the write end to the provider.
- 170 • If required, the provider may send additional information, such as a filter
171 to receive only a subset of the available records.
- 172 • The consumer may pause receiving data by not reading from the pipe. The
173 provider should add the pipe to its main loop in non-blocking mode; it
174 will receive write error `EAGAIN` if the pipe is full (paused). The provider
175 must be careful to write a whole record at a time: even if it received `EA-`
176 `GAIN` part way through a record and skipped subsequent records, it must
177 finish writing the partial record before doing anything else. Otherwise,
178 the structure of the stream is likely to be corrupted.
- 179 • If there are n providers, the consumer would read from n pipes, and could
180 receive new records from any of them.
- 181 • If there are m consumers, the provider would have m pipes, and would
182 normally write each new record into each of them.
- 183 • The consumer may stop receiving data by closing the pipe. The provider
184 will receive write error `EPIPE`, and should respond by also closing that

185 pipe.
 186 • If required, the consumer could control the provider by calling additional
 187 methods. For instance, the interface might define a `ChangeFilter()`
 188 method.

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

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

203 For the framing protocol, the serialization protocol might provide its own fram-
 204 ing (for example, fixed-length structures of a known length do not need framing),
 205 or the interface might document the use of an existing framing protocol such
 206 as [netstrings](https://en.wikipedia.org/wiki/Netstring)⁸, or its own framing/packetization protocol such as “4-byte little-
 207 endian length followed by that much data”.

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

213 A complete interface might look like this (pseudocode):

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

⁸<https://en.wikipedia.org/wiki/Netstring>

225 **Provider-initiated push via a stream**

226 If the initiator is the provider and the data/message rates make D-Bus unsuit-
227 able, the consumer could implement an interface that receives events from an
228 out-of-band stream that is provided by the provider when it initiates communi-
229 cation, again using the D-Bus type “h”(file-handle) for file descriptor passing.

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

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

248 A complete interface might look like this (pseudocode):

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

251 **Bidirectional communication via D-Bus**

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

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

259 **Bidirectional communication via a socket or pair of pipes**

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

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

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

270 Resuming communication

271 If the system is restarted and the previously running applications are restored,
272 and the interface is one where resuming communication makes sense, we rec-
273 ommend that the original initiator re-initiates communication. This would nor-
274 mally be done by repeating [interface discovery](#)⁹.

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

278 Stored state

279 In some interfaces, the provider has a particular state stored in-memory or
280 on-disk at any given time, and the inter-process communication works by pro-
281 viding enough information that the consumer can reproduce that state. This
282 approach is recommended, particularly for [publish/subscribe](#) interfaces, where
283 it is conventionally what is done.

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

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

295 In interfaces where the provider is stateless and has “[carousel](#)¹⁰”behaviour, the
296 consumer may cache past items/events in memory or on disk for as long as they
297 are considered valid.

298 Similarly, if a provider that receives items from a carousel implements an inter-
299 face that expects it to store state, the provider may cache past items/events in

⁹https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_framework/interface_discovery/

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

300 memory or on disk for as long as they are considered valid, so that they can be
301 provided to the consumer.