



Data sharing

1 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	5
10	Consumer-initiated pull via a stream	6
11	Provider-initiated push via a stream	7
12	Bidirectional communication via D-Bus	8
13	Bidirectional communication via a socket or pair of pipes	8
14	Resuming communication	8
15	Stored state	9

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

21 Use cases

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

27 Selecting an initiator

28 The first design question is which peer should initiate the connection (the *ini-*
29 *tiator*) and which one should not (the *responder*).

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

33 Some guidelines:

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

¹https://jwd.pages.apertis.org/apertis-website/concepts/points_of_interest/

²<https://jwd.pages.apertis.org/apertis-website/concepts/sharing/>

³<https://jwd.pages.apertis.org/apertis-website/concepts/global-search/>

⁴https://jwd.pages.apertis.org/apertis-website/concepts/interface_discovery/

- 37 • If one of the peers is assumed to be running already, but the other can
38 be auto-started on-demand, then the peer that is running already should
39 be the initiator, and the peer that can be auto-started should be the
40 responder.
- 41 • If the connection is normally only established when one of the peers re-
42 ceives user input, then that peer should be the initiator.
- 43 • If there is no other reason to prefer one direction over the other, the
44 consumer is usually the initiator.

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

48 Discovery

49 Each initiator carries out [Interface discovery](#)⁵ to find implementations of the
50 responder. If the initiator is the consumer, the interface that is discovered
51 might have a name like `com.example.PointsOfInterestProvider`. If the initia-
52 tor is the provider, the interface that is discovered might have a name like
53 `com.example.DebugLogConsumer`.

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

58 Connection

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

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

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

⁵https://jwd.pages.apertis.org/apertis-website/concepts/interface_discovery/

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

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

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

89 Communication

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

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

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

98 Publish/subscribe via D-Bus

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

- 102 • The consumer can receive the initial state of the provider by calling a
103 method such as `ListPointsOfInterest()`, or by retrieving its D-Bus proper-
104 ties using `GetAll()`. This method call is often referred to as *state recovery*.
- 105 • The provider can notify all consumers of changes to its state by emitting
106 broadcast signals, or notify a single consumer by using unicast signals.
107 The consumer is expected to connect D-Bus signal handlers *before* it calls
108 the initial method, to avoid missing events.
- 109 • We recommend that the provider should hold its state on disk or in mem-
110 ory so that it can provide state recovery. However, if there is a strong
111 reason for a particular interaction to use a “[carousel](#)⁷” model in which

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

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

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

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

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

128 Query-based access via D-Bus

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

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

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

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

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

143 Provider-initiated push via D-Bus

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

- 147 • the provider can send data by calling a method such as `AddPointsOfIn-`
148 `terest()`

- 149 • if required, the consumer can influence the provider(s) by emitting broad-
150 cast or unicast D-Bus signals defined by the interface (for example an inter-
151 face might define `PauseRequested`, `ResumeRequested` and/or `RefreshRe-`
152 `requested` signals)

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

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

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

159 Consumer-initiated pull via a stream

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

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

188 The advantages of this design are its high efficiency and low latency. The major
189 disadvantage of this design is that the provider and consumer need to agree

190 on a framing and serialization protocol with which they can write records into
191 the stream and read them out again. Designing the framing and serialization
192 protocol is part of the design of the interface.

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

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

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

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

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

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

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

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

- 229
- 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.

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

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

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

250 **Bidirectional communication via D-Bus**

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

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

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

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

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

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

269 **Resuming communication**

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

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

277 **Stored state**

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

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

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

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

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

⁹https://jwd.pages.apertis.org/apertis-website/concepts/interface_discovery/

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