# XPC and App-to-App Communication

This thread has been locked by a moderator.

112

I've explained this issue many times before, both here on DevForums and in DTS tech support incidents, but never in a coherent fashion. This week I received yet another DTS TSI about this issue, and I'm using that as an excuse to write it up properly (-:

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

## XPC and App-to-App Communication

There is no supported way to directly communicate between apps using XPC.

### In the beginning...

... there was Mach messaging.

In Mach messaging, services are represented by a **port**, a kernel object that manages message-based IPC. A **server** has a **receive right** for a port, allowing it to receive messages that were sent to that port. A **client** that wants to send a message to a port must have a **send right** for that port.

Mach is a capability-based system. You can't create a send right from scratch; you must be granted it by someone. Mach messages can transfer send rights from process to process. However, this presents a chicken and egg problem: How do you get your first send right?

The answer here is the **bootstrap service**. Every process starts with a send right to a **bootstrap port**. When a process wants to access a service, it sends a message with the service name to the bootstrap port. On success, the bootstrap service replies with a message with a send right to the port for that service.

Not all processes talk to the same bootstrap port. Rather, the system manages multiple bootstrap ports, where each port represents a **bootstrap namespace**. The system gives each process a send right to the bootstrap port that's appropriate for its execution context. The bootstrap service uses this bootstrap port to determine what service names are accessible to a client talking to that port.

These bootstrap namespaces form a tree. At the root there is a **global bootstrap namespace**. A `launchd` daemon runs in that namespace. Below that are a set of **per-user bootstrap namespaces**, and below those are **per-session bootstrap namespaces** for each login sessions. A GUI app runs in a login session namespace.

For a more in-depth explanation of this concept, see the *Execution Contexts* section of Technote 2083 Daemons and Agents.

**IMPORTANT** That technote is very old and the bootstrap namespace model is now significantly more complex than what's described there. However, the basic ideas are still valid.

### XPC Fundamentals

XPC wraps Mach messaging in an API that's *much* easier to use. An **XPC connection** represents a communication channel between two processes. An **XPC listener** listens for incoming connections. While there are anonymous listeners, most listeners are associated with a **named endpoint**, where the name is registered in a bootstrap namespace. This is what allows the client to connect to the listener by name.

XPC is tightly integrated with the on-demand architecture supported by `launchd`. `launchd` manages a set of **jobs** — XPC services, `launchd` daemons and agents, and so on — and each job publishes a set of named endpoints. Client processes connect to these endpoints by name. Under the covers, XPC looks up the name in the client's bootstrap namespace.

A `launchd` job doesn't need to run to publish its named endpoints. Rather, `launchd` learns about the endpoints by reading a property list associated with the job. For example:

- An XPC service advertises a single named endpoint, namely the bundle ID in the service's `Info.plist`; for the details, see the `xpcservice.plist` man page.
- A `launchd` daemon can advertise multiple named endpoints via the `MachServices` property in its `launchd` property list; for the details, see the `launchd.plist` man page.

`launchd` monitors these named endpoints for demand. When a client process sends a message to a connection that targets a named endpoint, `launchd` starts the associated job. The job then services the demand by starting XPC listeners for its named endpoints.

XPC has two APIs:

- The low-level C API
- The Foundation XPC API, commonly referred to by the main class name, `NSXPCConnection`

This post focuses on the latter but the same concepts apply to both.

### App-to-App Problems

The XPC architecture is incompatible with direct app-to-app communication:

- There's no way for `launchd` to know what named endpoints it should monitor on your app's behalf.
- Launching an app is a heavyweight operation, one clearly visible to the user, so it's not something that `launchd` can do on demand.

This limitation is reflected in the XPC API. Specifically, there are three ways to create an XPC listener:

- The `service()` class method — This creates a listener for an XPC service's named endpoint.
- The `init(machServiceName:)` initialiser — This creates a listener for one of the names advertised in the `MachServices` property of a `launchd` daemon or agent.
- The `anonymous()` class method — This creates an anonymous listener.

None of these are useful in setting up app-to-app communication.

### The Xcode Gotcha

One particularly gnarly gotcha here is that app-to-app communication using XPC works when you run your apps from Xcode. This is a side effect of the infrastructure used by Xcode to debug XPC services. That infrastructure allows the listener app to create a listener using `init(machServiceName:)` even though the corresponding service name is not known to `launchd`. So your code works in the debugger but then fails when you run it from the Finder. Ouch!

### Alternatives

If you can't use XPC for app-to-app communication, what are the alternatives? Here's a short list of things that might work:

- Unix domain sockets — For the details, see the `unix` man page, or any good text book an BSD Sockets.
- `CFMessagePort` — For the details, see its documentation.
- XPC rendezvous — See the *XPC Rendezvous* section, below.

Which is best depends on your circumstances. Unix domain sockets is an industry standard API that works well. It relies on the BSD Sockets API, which is un-fun to call from Swift. Its access control is based on file system permissions, which is helpful if you need to cut across bootstrap namespaces.

In contrast, `CFMessagePort` is a thin wrapper around Mach messaging. That means that its tied to your bootstrap namespace, which can be useful. It's relatively easy to call from Swift, but still not trivial.

XPC rendezvous is based on XPC, so it has all of its advantages. The main disadvantages is that it requires a `launchd` job to help with the rendezvous, which isn't always feasible.

Oh, and here's a short list of things to avoid:

- Mach messaging — I *strongly* recommend against using Mach messaging directly. It's almost impossible to use correctly.
- Distributed Objects (DO) — This has been deprecated for many years now, and for good reason. It has a wide range of weird and wonderful bugs.

### XPC Rendezvous

One way to set up app-to-app communication is with an XPC rendezvous. This technique requires a `launchd` job that's visible to both parties:

1. This `launchd` job advertises a named endpoint.
2. Client A calls the `anonymous()` class method to create an anonymous listener.
3. It then uses the `endpoint` property to get an endpoint (`NSXPCListenerEndpoint`) for that listener.
4. It uses XPC to send this endpoint to the `launchd` job.
5. The `launchd` job stores this endpoint.
6. Client B uses XPC to get the endpoint from the `launchd` job.
7. Client B passes the endpoint to the `init(listenerEndpoint:)` initialiser to open a connection directly to client A.

**IMPORTANT** The `launchd` job in step one cannot be an XPC service. Third-party XPC services are always scoped to their container app (see the discussion of the `ServiceType` property in the `xpcservice.plist` man page) and thus can't fulfil the primary requirement of an XPC rendezvous, namely, to be visible to both parties.

Most other `launchd` jobs do work for this, including:

- `launchd` daemons and agents
- Service Management login items
- System extensions

XPC

Reply

Posted 3 weeks ago by eskimo

Add a Comment