Creating XPC Services

The XPC Services API, part of libSystem, provides a lightweight mechanism for basic interprocess communication integrated with Grand Central Dispatch (GCD) and `launchd`. The XPC Services API allows you to create lightweight helper tools, called XPC services, that perform work on behalf of your application.

There are two main reasons to use XPC services: privilege separation and stability.

Stability:

Let's face it; applications sometimes crash. We don't want it to happen, but it does anyway. Often, certain parts of an application are more prone to crashes than others. For example, the stability of any application with a plug-in API is inherently at the mercy of the authors of plug-ins.

When one part of an application is more at risk for crashes, it can be useful to separate out the potentially unstable functionality from the core of the application. This separation lets you ensure that that, if it crashes, the rest of the application is not affected.

Privilege Separation:

Modern applications increasingly rely on untrusted data, such as web pages, files sent by email, and so on. This represents a growing attack vector for viruses and other malware.

With traditional applications, if an application becomes compromised through a buffer overflow or other security vulnerability, the attacker gains the ability to do anything that the user can do. To mitigate this risk, Mac OS X provides sandboxing—limiting what types of operations a process can perform.

In a sandboxed environment, you can further increase security with privilege separation—dividing an application into smaller pieces that are responsible for a part of the application's behavior. This allows each piece to have a more restrictive sandbox than the application as a whole would require.

Other mechanisms for dividing an application into smaller parts, such as `NSTask` and `posix_spawn`, do not let you put each part of the application in its own sandbox, so it is not possible to use them to implement privilege separation. Each XPC service has its own sandbox, so XPC services can make it easier to implement proper privilege separation.

For more information about sandboxing, see *App Sandbox Design Guide*.

Understanding the Structure and Behavior

XPC services are managed by `launchd`, which launches them on demand, restarts them if they crash, and terminates them (by sending `SIGKILL`) when they are idle. This is transparent to the application using the service, except for the case of a service that crashes while processing a message that requires a response. In that case, the application can see that its XPC connection has become invalid until the service is restarted by `launchd`. Because an XPC service can be terminated suddenly at any time, it must be designed to hold on to minimal state—ideally, your service should be completely stateless, although this is not always possible.

By default, XPC services are run in the most restricted environment possible—sandboxed with minimal filesystem access, network access, and so on. Elevating a service's privileges to root is not supported. Further, an XPC service is private, and is available only to the main application that contains it.

Choosing an XPC API

Beginning in Mac OS X v10.8, there are two different APIs for working with XPC services: the XPC Services API and the `NSXPCConnection` API.

The `NSXPCConnection` API is an Objective-C-based API that provides a remote procedure call mechanism, allowing the client application to call methods on proxy objects that transparently relay those calls to corresponding objects in the service helper and vice-versa.

The XPC Services API is a C-based API that provides basic messaging services between a client application and a service helper.

The XPC Services API is recommended if you need compatibility with Mac OS X v10.7, or if your application and its service are not based on the Foundation framework. The `NSXPCConnection` API is recommended for apps based on the Foundation framework in Mac OS X v10.8 and later.

The NSXPCConnection API

The `NSXPCConnection` API is part of the Foundation framework, and is described in the `NSXPCConnection.h` header file. It consists of the following pieces:

- `NSXPCConnection`—a class that represents the bidirectional communication channel between two processes. Both the application and the service helper have at least one connection object.
- `NSXPCInterface`—a class that describes the expected programmatic behavior of the connection (what classes can be transmitted across the connection, what objects are exposed, and so on).
- `NSXPCListener`—a class that listens for incoming XPC connections. Your service helper must create one of these and assign it a delegate object that conforms to the `NSXPCListenerDelegate` protocol.
- `NSXPCListenerEndpoint`—a class that uniquely identifies an `NSXPCListener` instance and can be sent to a remote process using `NSXPCConnection`. This allows a process to construct a direct communication channel between two other processes that otherwise could not see one another.

In addition, the header contains a few constants, described in *NSXPCConnection Constants Reference*.

The XPC Services API

The (C-based) XPC Services API consists of two main pieces:

- `xpc.h`—APIs for creating and manipulating property list objects and APIs that daemons use to reply to messages.

  Because this API is at the libSystem level, it cannot depend on Core Foundation. Also, not all CF types can be readily shared across process boundaries. XPC supports connection types such as file descriptors in its object graphs, which are not supported by CFPropertyList because it was designed as a persistent storage format, whereas XPC was designed as an IPC format. For these reasons, the XPC API uses its own container that supports only the primitives that are practical to transport across process boundaries.

  Some higher-level types can be passed across an XPC connection, although they do not appear in the `xpc.h` header file (because referencing higher level frameworks from libSystem would be a layering violation). For example, the `IOSurfaceCreateXPCObject` and `IOSurfaceLookupFromXPCObject` functions allow you to pass an IOSurface object between the XPC service that does the drawing and the main application.

- `connection.h`—APIs for connecting to an XPC service. This service is a special helper bundle embedded in your app bundle.

  A connection is a virtual endpoint; it is independent of whether an actual instance of the service binary is running. The service binary is launched on demand.

  A connection can also be sent as a piece of data in an XPC message. Thus, you can pass a connection through XPC to allow one service to communicate with another service (for example).

Note: The underlying encoding used by XPC is opaque to the user, and so is the communication channel. You should not attempt to interact directly with either, as they are subject to change without notice.

You should also not try to archive the bytes of a message or the objects that contain it to disk; the encoding is not considered an ABI contract, and may change at any time.

Creating the Service

An XPC service is a bundle in the `Contents/XPCServices` directory of the main application bundle; the XPC service bundle contains an `Info.plist` file, an executable, and any resources needed by the service. The XPC

service indicates which function to call when the service receives messages by calling `xpc_main(3) Mac OS X Developer Tools Manual Page` from its `main` function.

To create an XPC service in Xcode, do the following:

1. Add a new target to your project, using the XPC Service template.
2. Add a Copy Files phase to your application's build settings, which copies the XPC service into the `Contents/XPCServices` directory of the main application bundle.
3. Add a dependency to your application's build settings, to indicate it depends on the XPC service bundle.
4. If you are writing a low-level (C-based) XPC service, implement a minimal `main` function to register your event handler, as shown in the following code listing. Replace `my_event_handler` with the name of your event handler function.

```
int main(int argc, const char *argv[]) {
    xpc_main(my_event_handler);

    // The xpc_main() function never returns.
    exit(EXIT_FAILURE);
}
```

   If you are writing a high-level (Objective-C-based) service using `NSXPCConnection`, first create a connection delegate class that conforms to the `NSXPCListenerDelegate` protocol. Then, implement a minimal `main` function that creates and configures a listener object, as shown in the following code listing.

```
int main(int argc, const char *argv[]) {
    MyDelegateClass *myDelegate = ...
    NSXPCListener *listener =
        [NSXPCListener serviceListener];

    listener.delegate = myDelegate;
    [listener resume];

    // The resume method never returns.
    exit(EXIT_FAILURE);
}
```

   The details of this class are described further in Using the Service.
5. Add the appropriate key/value pairs to the helper's `Info.plist` to tell launchd the name of the service. These are described in XPC Service Property List Keys.

Note: If you want to sandbox your service, it must also be signed; entitlements are stored as part of the code signature.

Using the Service

The way you use an XPC service depends on whether you are working with the C API (XPC Services) or the Objective-C API (`NSXPCConnection`).

Using the Objective-C NSXPCConnection API

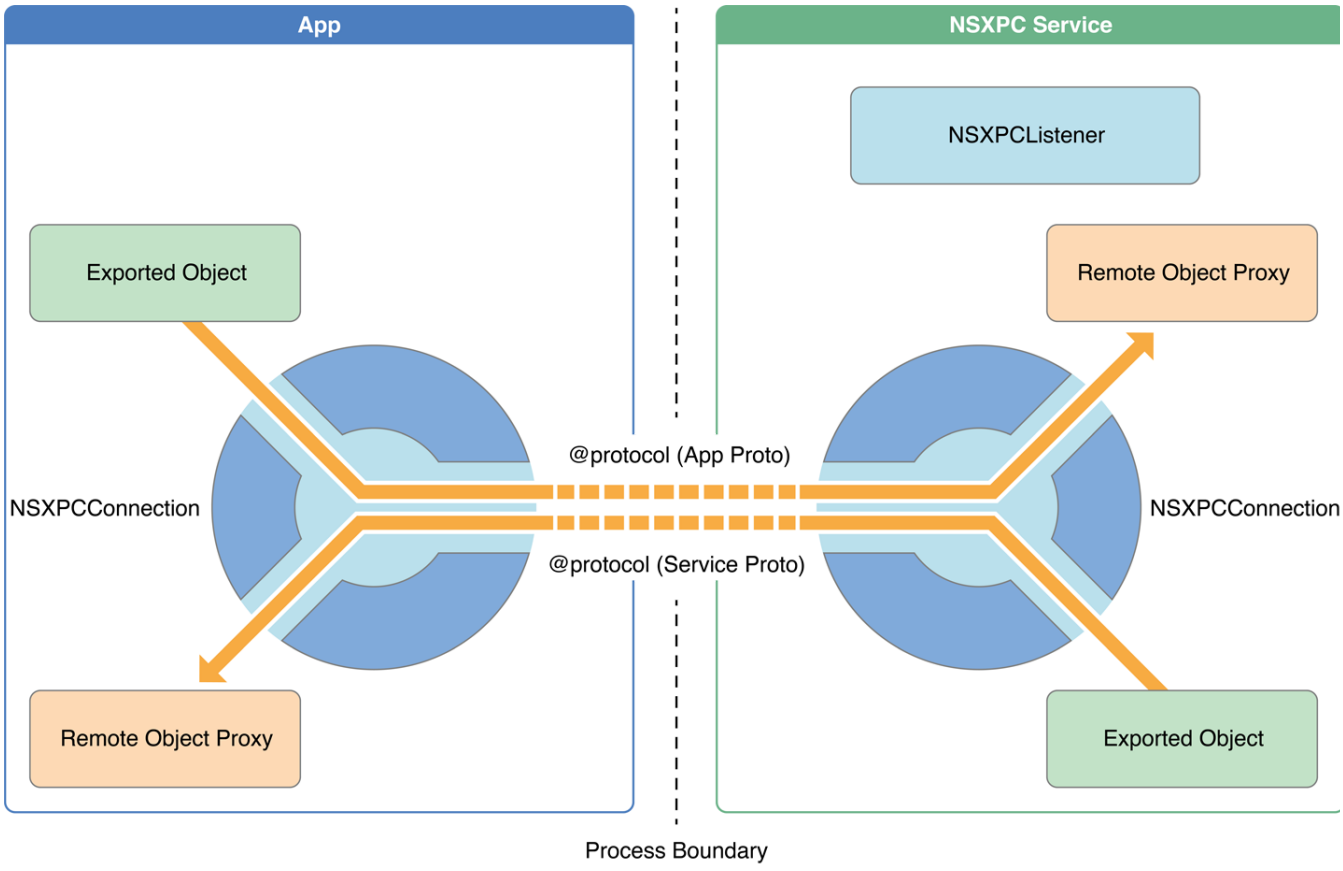
The Objective-C `NSXPCConnection` API provides a high-level remote procedure call interface that allows you to call methods on objects in one process from another process (usually an application calling a method in an XPC service). The `NSXPCConnection` API automatically serializes data structures and objects for transmission and deserializes them on the other end. As a result, calling a method on a remote object behaves much like calling a method on a local object.

To use the `NSXPCConnection` API, you must create the following:

- An interface. This mainly consists of a protocol that describes what methods should be callable from the remote process. This is described in Designing an Interface
- A connection object on both sides. On the service side, this was described previously in Creating the Service. On the client side, this is described in Connecting to and Using an Interface.
- A listener. This code in the XPC service accepts connections. This is described in Accepting a Connection in the Helper.
- Messages.

Figure 4-1 shows how these pieces fit together.

Figure 4-1 The NSXPC architecture



In some cases, you may need to further tweak the protocol to whitelist additional classes for use in collection parameters or to proxy certain objects instead of copying them. This is described further in Working with Custom Classes.

Overall Architecture

When working with `NSXPCConnection`-based helper apps, both the main application and the helper have an instance of `NSXPCConnection`. The main application creates its connection object itself, which causes the helper to launch. A delegate method in the helper gets passed its connection object when the connection is established. This is illustrated in Figure 4-1.

Each `NSXPCConnection` object provides three key features:

- An `exportedInterface` property that describes the methods that should be made available to the opposite side of the connection.
- An `exportedObject` property that contains a local object to handle method calls coming in from the other side of the connection.
- The ability to obtain a proxy object for calling methods on the other side of the connection.

When the main application calls a method on a proxy object, the XPC service's `NSXPCConnection` object calls that method on the object stored in its `exportedObject` property.

Similarly, if the XPC service obtains a proxy object and calls a method on that object, the main app's `NSXPCConnection` object calls that method on the object stored in its `exportedObject` property.

Designing an Interface

The `NSXPCConnection` API takes advantage of Objective-C protocols to define the programmatic interface between the calling application and the service. Any instance method that you want to call from the opposite side of a connection must be explicitly defined in a formal protocol. For example:

```
@protocol FeedMeACookie
    - (void)feedMeACookie: (Cookie *)cookie;
@end
```

Because communication over XPC is asynchronous, all methods in the protocol must have a return type of `void`. If you need to return data, you can define a reply block like this:

```
@protocol FeedMeAWatermelon
    - (void)feedMeAWatermelon: (Watermelon *)watermelon
        reply:(void (^)(Rind *))reply;
@end
```

A method can have only one reply block. However, because connections are bidirectional, the XPC service helper can also reply by calling methods in the interface provided by the main application, if desired.

Each method must have a return type of `void`, and all parameters to methods or reply blocks must be either:

- Arithmetic types (`int`, `char`, `float`, `double`, `uint64_t`, `NSUInteger`, and so on)
- `BOOL`
- C strings
- C structures and arrays containing only the types listed above
- Objective-C objects that implement the `NSSecureCoding` protocol.

Important: If a method (or its reply block) has parameters that are Objective-C collection classes (`NSDictionary`, `NSArray`, and so on), and if you need to pass your own custom objects within a collection, you must explicitly tell XPC to allow that class as a member of that collection parameter. For details, read Working with Custom Classes.

Connecting to and Using an Interface

Once you have defined the protocol, you must create an interface object that describes it. To do this, call the `interfaceWithProtocol:` method on the `NSXPCInterface` class. For example:

```
NSXPCInterface *myCookieInterface =
```

```
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedMeACookie)];
```

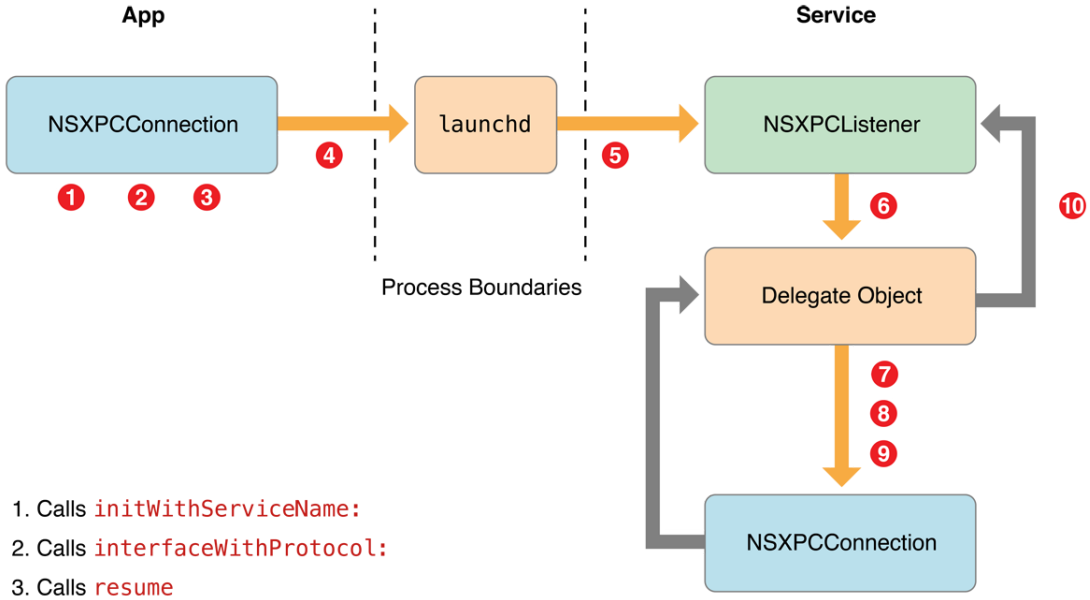
Once you have created the interface object, within the main app, you must configure a connection with it by calling the `initWithServiceName:` method. For example:

```
NSXPCConnection *myConnection =    [[NSXPCConnection alloc]
     initWithServiceName:@"com.example.monster"];
myConnection.remoteObjectInterface = myCookieInterface;
[myConnection resume];
```

Note: For communicating with XPC services outside your app bundle, you can also configure an XPC connection with the `initWithMachServiceName:` method. For details, see the documentation for that method.

Figure 4-2 shows the basic steps in this process. Note that only the first four steps are described in this section.

Figure 4-2 The NSXPC connection process



1. Calls `initWithServiceName:`
2. Calls `interfaceWithProtocol:`
3. Calls `resume`
4. Connects to `launchd`
5. `launchd` starts service
6. Listener object calls `listener:shouldAcceptNewConnection:` on delegate
7. Sets `exportedObject`
8. Sets `exportedInterface`
9. Calls `resume`
10. Returns `YES`

At this point, the main application can call the `remoteObjectProxy` or `remoteObjectProxyWithErrorHandler:` methods on the `myConnection` object to obtain a proxy object. This object acts as a proxy for the object that the XPC service has set as its exported object (by setting the `exportedObject` property). This object must conform to the protocol defined by the `remoteObjectInterface` property.

When your application calls a method on the proxy object, the corresponding method is called on the exported object inside the XPC service. When the service's method calls the reply block, the parameter values are serialized and sent back to the application, where the parameter values are deserialized and passed to the reply block. (The reply block executes within the application's address space.)

Note: If you want to allow the helper process to call methods on an object in your application, you must set the `exportedInterface` and `exportedObject` properties before calling `resume`. These properties are described further in the next section.

Accepting a Connection in the Helper

As shown in Figure 4-2, when an `NSXPCConnection`-based helper receives the first message from a connection, the listener delegate's `listener:shouldAcceptNewConnection:` method is called with a listener object and a connection object. This method lets you decide whether to accept the connection or not; it should return `YES` to accept the connection or `NO` to refuse the connection.

Note: The helper receives a connection request when the first actual message is sent. The connection object's `resume` method does not cause a message to be sent.

In addition to making policy decisions, this method must configure the connection object. In particular, assuming the helper decides to accept the connection, it must set the following properties on the connection:

- `exportedInterface`—an interface object that describes the protocol for the object you want to export. (Creating this object was described previously in Connecting to and Using an Interface.)
- `exportedObject`—the local object (usually in the helper) to which the remote client's method calls should be delivered. Whenever the opposite end of the connection (usually in the application) calls a method on the connection's proxy object, the corresponding method is called on the object specified by the `exportedObject` property.

After setting those properties, it should call the connection object's `resume` method before returning `YES`. Although the delegate may defer calling `resume`, the connection will not receive any messages until it does so.

Sending Messages

Sending messages with NSXPC is as simple as making a method call. For example, given the interface `myCookieInterface` (described in previous sections) on the XPC connection object `myConnection`, you can call the `feedMeACookie` method like this:

```
Cookie *myCookie = ...


[[myConnection remoteObjectProxy] feedMeACookie: myCookie];
```

When you call that method, the corresponding method in the XPC helper is called automatically. That method, in turn, could use the XPC helper's connection object similarly to call a method on the object exported by the main application.

Handling Errors

In addition to any error handling methods specific to a given helper's task, both the XPC service and the main app should also provide the following XPC error handler blocks:

- Interruption handler—called when the process on the other end of the connection has crashed or has otherwise closed its connection.

  The local connection object is typically still valid—any future call will automatically spawn a new helper instance unless it is impossible to do so—but you may need to reset any state that the helper would otherwise have kept.

  The handler is invoked on the same queue as reply messages and other handlers, and it is always executed after any other messages or reply block handlers (except for the invalidation handler). It is safe to make new requests on the connection from an interruption handler.

- Invalidation handler—called when the invalidate method is called or when an XPC helper could not be started. When this handler is called, the local connection object is no longer valid and must be recreated.

  This is always the last handler called on a connection object. When this block is called, the connection object has been torn down. It is not possible to send further messages on the connection at that point, whether inside the handler or elsewhere in your code.

In both cases, you should use block-scoped variables to provide enough contextual information—perhaps a pending operation queue and the connection object itself—so that your handler code can do something sensible, such as retrying pending operations, tearing down the connection, displaying an error dialog, or whatever other actions make sense in your particular app.

Working with Custom Classes

Note: Before you read this section, you should read the chapters Serializations and Serializing Property Lists in *Archives and Serializations Programming Guide* to learn the basics of object serialization in Mac OS X.

The `NSXPCConnection` class limits what objects can be passed over a connection. By default, it allows only known-safe classes—Foundation collection classes, `NSString`, and so on. You can identify these classes by whether they conform to the `NSSecureCoding` protocol.

Only classes that conform to this protocol can be sent to an `NSXPCConnection`-based helper. If you need to pass your own classes as parameters, you must ensure that they conform to the `NSSecureCoding` protocol, as described below.

However, this is not always sufficient. You need to do extra work in two situations:

- If you are passing the object inside a collection (dictionary, array, and so on).
- If you need to pass the object by proxy instead of copying the object.

All three cases are described in the sections that follow.

Conforming to NSSecureCoding

All objects passed over an NSXPC connection must conform to `NSSecureCoding`. to do this, your class must do the following:

- **Declare support for secure coding.** Override the `supportsSecureCoding` method, and make it return `YES`.
- **Decode singleton class instances safely.** If the class overrides its `initWithCoder:` method, when decoding any instance variable, property, or other value that contains an object of a non-collection class (including custom classes) always use `decodeObjectOfClass:forKey:` to ensure that the data is of the expected type.
- **Decode collection classes safely.** Any non-collection class that contains instances of collection classes *must* override the `initWithCoder:` method. In that method, when decoding the collection object or objects, always use `decodeObjectOfClasses:forKey:` and provide a list of any objects that can appear within the collection.

When generating the list of classes to allow within a decoded collection class, you should be aware of two things.

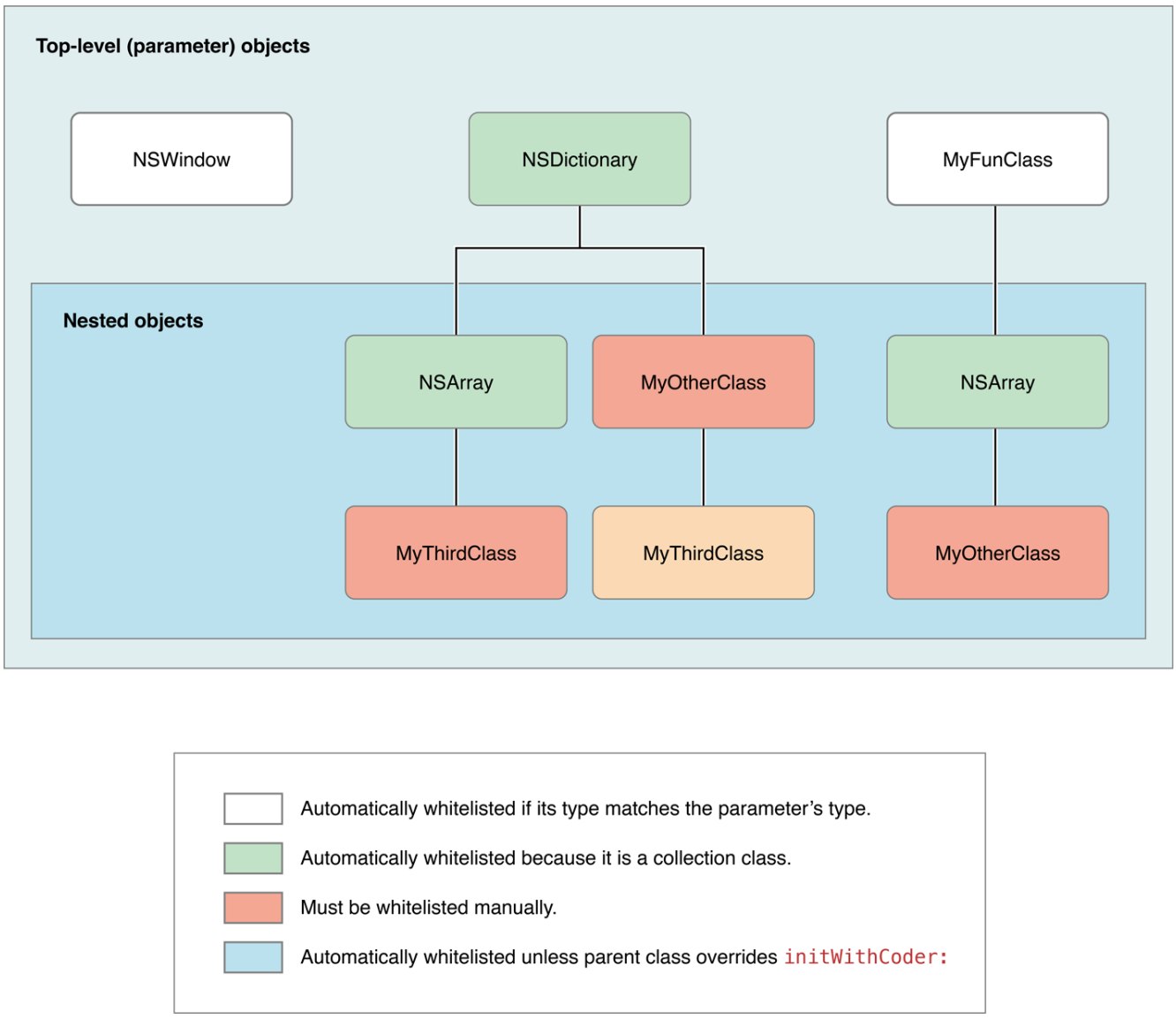
First, Apple collection classes are *not* automatically whitelisted by the `decodeObjectOfClasses:forKey:` method, so you must include them explicitly in the array of class types.

Second, you should list only classes that are direct members of the collection object graph that you are decoding without any intervening non-collection classes.

For example, if you have an array of dictionaries, and one of those dictionaries might contain an instance of a custom class called `OuterClass`, and `OuterClass` has an instance variable of type InnerClass, you must include `OuterClass` in the list of classes because it is a direct member of the collection tree. However, you do not need to list `InnerClass` because there is a non-collection object between it and the collection tree.

Figure 4-3 shows some examples of when whitelisting is required and shows when classes must provide overridden `initWithCoder`: methods.

Figure 4-3 Whitelisting and secure coding examples



Whitelisting a Class for Use Inside Containers

Most of the time, custom classes passed through method parameters can be automatically whitelisted based upon the method's signature. However, when a method specifies a collection class (`NSArray`, `NSDictionary`, and so on) as the parameter type, the compiler has no way to determine what classes should be allowed to appear within that container. For this reason, if your methods take collection class instances as parameters, you must explicitly whitelist any classes that can appear within those containers.

For every method in your interface that takes a collection class as a parameter, you must determine what classes should be allowed as members.

You should whitelist *only* classes that can be members of any top-level collection objects. If you whitelist classes at the top level unnecessarily, those objects are allowed to appear within the top-level collection objects, which is not what you want. In particular:

- Apple-provided classes that support property list serialization (such as other collection classes) are automatically whitelisted for you. It is never necessary to whitelist these classes at the top level.

  For the most up-to-date list of classes supported by property list serialization, read Serializing Property Lists in *Archives and Serializations Programming Guide*.

- Don't whitelist custom classes if they are allowed only inside instances of other custom classes. If the enclosing class correctly conforms to `NSSecureCoding`, such whitelisting is not required. For more details, read Conforming to NSSecureCoding.

For example, suppose you have the following class and protocol:

```
@interface FrenchFry
...
@end

@protocol FeedMeABurgerAndFries
    - (void)feedMeFries: (NSArray *)pommesFrites;
    - (void)feedMeABurger: (Burger *)aBurger;
@end
```

Assuming the `pommesFrites` array is an array of `FrenchFry` objects (as the name implies), you would whitelist the array of `FrenchFry` objects as follows:

```
// Create the interface
NSXPCInterface *myBurgerInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedMeABurgerAndFries)];

// Create a set containing the allowed
// classes.
NSSet *expectedClasses =
    [NSSet setWithObjects:[NSArray class], [FrenchFry class], nil];

[myBurgerInterface
      setClasses: expectedClasses
     forSelector: @selector(feedMeFries:)
   argumentIndex: 0  // the first parameter
         ofReply: NO // in the method itself.
];
```

The first parameter to a method is parameter 0, followed by parameter 1, and so on.

In this case, the value `NO` is passed for the `ofReply` parameter because this code is modifying the whitelist for one of the parameters of the method itself. If you are whitelisting a class for a parameter of the method's reply block, pass `YES` instead.

Passing an Object By Proxy

Most of the time, it makes sense to copy objects and send them to the other side of a connection. Passing objects by copying is the most straightforward way to use NSXPC, and should be used wherever possible.

However, copying objects is not always desirable. In particular:

- If you need to share a single instance of the data between the client application and the helper, you must pass the objects by proxy.
- If an object needs to call methods on other objects within your application that you cannot or do not wish to pass across the connection (such as user interface objects), then you must pass an object by proxy—either the caller, the callee (where possible), or a relay object that you construct specifically for that purpose.

The downside to passing objects by proxy is that performance is significantly reduced (because every access to the object requires interprocess communication). For this reason, you should pass objects by proxy *only* if it is not possible to pass them by copying.

You can configure additional proxy objects similarly to the way you configured the `remoteObjectInterface` property of the initial connection. First, identify which parameter to a method should be passed by proxy, then specify an `NSXPCInterface` object that defines the interface for that object.

Suppose, for example, that you have a class conforming to the following protocol:

```
@protocol FeedSomeone
    - (void)feedSomeone:
        (id <FeedMeACookie>)someone;
@end

...

NSXPCInterface *myFeedingInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedSomeone)];
```

If you want to pass the first parameter to that method by proxy, you would configure the interface like this:

```
// Create an interface object that describes
// the protocol for the object you want to
// pass by proxy.
NSXPCInterface *myCookieInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedMeACookie)];

// Create an object of a class that
// conforms to the FeedMeACookie protocol
Monster *myMonster = ...

[myFeedingInterface
     setInterface: myCookieInterface
      forSelector: @selector(sendOtherProxy:)
    argumentIndex: 0  // the first parameter of
          ofReply: NO // the feedSomeone: method
];
```

The first parameter to a method is parameter 0, followed by parameter 1, and so on.

In this case, the value `NO` is passed for the `ofReply` parameter because this code is modifying the whitelist for one of the parameters of the method itself. If you are whitelisting a class for a parameter of the method's reply block, pass `YES` instead.

Using the C XPC Services API

Typical program flow is as follows:

1. An application calls `xpc_connection_create(3) Mac OS X Developer Tools Manual Page` to create an an XPC connection object.
2. The application calls some combination of `xpc_connection_set_event_handler(3) Mac OS X Developer Tools Manual Page` or `xpc_connection_set_target_queue(3) Mac OS X Developer Tools Manual Page` as needed to configure connection parameters prior to actually connecting to the service.

3. The application calls `xpc_connection_resume(3) Mac OS X Developer Tools Manual Page` to begin communication.
4. The application sends messages to the service using `xpc_connection_send_message(3) Mac OS X Developer Tools Manual Page`, `xpc_connection_send_message_with_reply(3) Mac OS X Developer Tools Manual Page`, or `xpc_connection_send_message_with_reply_sync(3) Mac OS X Developer Tools Manual Page`.
5. When you send the first message, the `launchd` daemon searches your application bundle for a service bundle whose `CFBundleIdentifier` value matches the specified name, then launches that XPC service daemon on demand.
6. The event handler function (specified in the service's `Info.plist` file) is called with the message. The event handler function runs on a queue whose name is the name of the XPC service.
7. If the original message was sent using `xpc_connection_send_message_with_reply(3) Mac OS X Developer Tools Manual Page` or `xpc_connection_send_message_with_reply_sync(3) Mac OS X Developer Tools Manual Page`, the service must reply using `xpc_dictionary_create_reply(3) Mac OS X Developer Tools Manual Page`, then uses `xpc_dictionary_get_remote_connection(3) Mac OS X Developer Tools Manual Page` to obtain the client connection and `xpc_connection_send_message(3) Mac OS X Developer Tools Manual Page`, `xpc_connection_send_message_with_reply(3) Mac OS X Developer Tools Manual Page`, or `xpc_connection_send_message_with_reply_sync(3) Mac OS X Developer Tools Manual Page` to send the reply dictionary back to the application.

   The service can also send a message directly to the application with `xpc_connection_send_message(3) Mac OS X Developer Tools Manual Page`.
8. If a reply was sent by the service, the handler associated with the previous message is called upon receiving the reply. The reply can be put on a different queue than the one used for incoming messages. No serial relationship is guaranteed between reply messages and non-reply messages.
9. If an error occurs (such as the connection closing), the connection's event handler (set by a previous call to `xpc_connection_set_event_handler(3) Mac OS X Developer Tools Manual Page`) is called with an appropriate error, as are (in no particular order) the handlers for any outstanding messages that are still awaiting replies.
10. At any time, the application can call `xpc_connection_suspend(3) Mac OS X Developer Tools Manual Page` when it needs to suspend callbacks from the service. All suspend calls must be balanced with resume calls. It is not safe to release the last reference to a suspended connection.
11. Eventually, the application calls `xpc_connection_cancel(3) Mac OS X Developer Tools Manual Page` to terminate the connection.

    Note: Either side of the connection can call `xpc_connection_cancel(3) Mac OS X Developer Tools Manual Page`. There is no functional difference between the application canceling the connection and the service canceling the connection.

XPC Service Property List Keys

XPC requires you to specify a number of special key-value pairs in the `Info.plist` file within the service helper's bundle. These keys are listed below.

`CFBundleIdentifier`
String. The name of the service in reverse-DNS style (for example, `com.example.myapp.myservice`). (This value is filled in by the XPC Service template.)

`CFBundlePackageType`
String. Value must be `XPC!` to identify the bundle as an XPC service. (This value is filled in by the XPC Service template.)

`XPCService`
Dictionary. Contains the following keys:

Key	Value
`EnvironmentVariables`	Dictionary. The variables which are set in the environment of the service.

`JoinExistingSession`	Boolean. Indicates that your service runs in the same security session as the caller. The default value is `False`, which indicates that the service is run in a new security session. Set the value to `True` if the service needs to access to the user's keychain, the pasteboard, or other per-session resources and services.
`RunLoopType`	String. Indicates the type of run loop used for the service. The default value is `dispatch_main`, which uses the `dispatch_main(3) Mac OS X Developer Tools Manual Page` function to set up a GCD-style run loop. The other supported value is `NSRunLoop`, which uses the `NSRunLoop` class to set up a run loop.