Network Interface Techniques

This thread has been locked by a moderator.

31

For important background information, read Extra-ordinary Networking before reading this.

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

Network Interface Techniques

Most developers don’t care about how the system manages network interfaces and their addresses. However, for some apps these details matter. This post is a high-level overview of some of the more common scenarios.

Before you read this, read Network Interface Concepts.

Binding to an Interface

Our networking APIs let you bind network operations to a specific interface. This triggers an effect known as *scoped routing*, where the route used by that operation’s traffic is determined by both the combination of its *source* and destination.

There are two ways this can happen:

- Explicitly, before starting the operation
- Implicitly, as the operation starts

The way you explicitly bind a network operation to an interface varies by API:

- With `URLSession` you set up constraints using properties on the `URLSessionConfiguration` object you use to create the session. For example, the `allowsCellularAccess` property ensures that operations done by the session will never run over WWAN.
- With Network framework you set up constraints using properties on the `NWParameters` object you use to create the `NWConnection`. For example, the `requiredInterface` property forces the connection to run over a specific interface.
- With BSD Sockets, the best way to bind a socket to an interface is with the `IP_BOUND_IF` (IPv4) or `IPV6_BOUND_IF` (IPv6) socket options.

A network operation can also be bound to an interface implicitly. This is easiest to understand with TCP. A TCP connection is uniquely identified by the local IP / local port / remote IP / remote port tuple [1]. When the system establishes a TCP connection, it decides on a local address. Once it’s done that, the local address can’t change because it’s part of the tuple that literally defines the connection. So, once you make a connection over a specific interface, it continues to run over that interface.

Note The connection is bound to the address, not the interface. If the interface’s address changes — for example, if the DHCP server refuses to renew the device’s lease and instead allocates it a new address — the connection will no longer function even though the interface is still available.

UDP does not use connections but Apple platforms have a similar concept known as a *UDP flow*, that is, all datagrams with the same local IP / local port / remote IP / remote port tuple. This concept shows up in a number of places, including:

- Network framework — When you use `NWConnection` with UDP, that represents a UDP flow.
- BSD Sockets — If you create a *connected* UDP socket, that represents a UDP flow.

UDP flows have the same behaviour as TCP connections when it comes to interface address changes.

[1] There’s a technology called Multipath TCP that lifts this invariant. Apple platforms lets you opt in to Multipath TCP, assuming your server supports it.

A Better Path

Consider this scenario:

1. Your app is on an iPhone that’s not connected to Wi-Fi.
2. It establishes a long-lived TCP connection to a server.
3. The iPhone joins a Wi-Fi network, and so the default route changes to Wi-Fi.

When happens to that connection?

It turns out that, due to scoped routing, the connection continues to work just fine. When you establish the connection, WWAN is the default route and so the connection’s local address is set to that of WWAN. That implicitly binds it to WWAN, so it keeps working regardless of the default route change in step 3.

This is both a good and bad thing. It’s good because the connection keeps working. It’s bad because the user might download a huge amount of data thinking that they’re on Wi-Fi.

If you’re building an app like this, check out the `betterPathUpdateHandler` property in Network framework. This is called when a better path is available. You can then:

1. Start up a new connection, which will use the better path.
2. Once that’s in place, gracefully close the old connection.

Constraints

The above example illustrates an important point: The network state can change at any time and it’s hard to stay in sync. Imagine you plan to use a connection to download a large resource. If you use a reactive approach, as described above, you might end up accidentally doing that over WWAN.

A better way to handle this is to declare constraints on your network operations. In Network framework, for example, you can set things like the `prohibitConstrainedPaths` property and the `prohibitExpensivePaths` property in the `NWParameters` object you use to create your connection. The system enforces those constraints for you, so you don’t have to write extra code to respond to network state changes.

These properties have a second benefit: The user may have a different definition of *expensive* than you. For example:

- Some users have unlimited WWAN data plans and are never on Wi-Fi, so WWAN is not expensive.
- The user’s iPad might be connected to a Personal Hotspot published by their iPhone, making Wi-Fi expensive.

If you use these constraints, your app will automatically honour the user’s settings, just like Apple’s built-in apps.

`URLSession` has similar facilities in the `URLSessionConfiguration` object, namely the `allowsConstrainedNetworkAccess` property and the `allowsExpensiveNetworkAccess` property. [Sorry that the names are inverted relative to Network framework, but the different polarities make sense in the context of the containing object.]

The documentation for the `URLSession` properties is currently much better than that for the Network framework properties, so start there even if you’re using Network framework. And for a general introduction to these constraints, watch WWDC 2019 Session 712 Advances in Networking, Part 1.

Network

Reply

Posted 2 days ago by eskimo

Add a Comment