Platform-Specific Networking Technologies

Networking in iOS and OS X are very similar, and most networking apps require little or no platform-specific code. However, you should be aware of a few small differences.

On iOS, you can use platform-specific networking APIs to handle authentication for captive networks and to designate Voice over Internet Protocol (VoIP) network streams. Also, iOS networking apps are much more likely to run on multihomed devices (usually with cellular and Wi-Fi connections), and must properly clean up network connections when the apps are put into the background.

The networking environment on OS X is highly configurable and extensible. The System Configuration framework provides APIs for determining and setting the current network configuration. Additionally, network kernel extensions enable you to extend the core networking infrastructure of OS X by adding features such as a firewall or VPN.

This chapter describes these platform-specific differences.

iOS Requires You to Handle Backgrounding and Specify Cellular Usage Policies

This section describes networking technologies and techniques that are specific to iOS, including information about captive network support, backgrounding, and making Wi-Fi-only connections.

Restrict Cellular Networking Correctly

There are two ways to prevent connections from being sent over a cellular network. Which method you use depends on your app's requirements and goals.

The `kSCNetworkReachabilityFlagsIsWWAN` flag in the `SCNetworkReachability` API tells you which interface will *probably* be used if your app connects to the specified host. However, this flag can be misleading because:

- The Wi-Fi signal could disappear after your app checks reachability, but before it connects.
- Different hosts may be reachable over different interfaces. You cannot trust a reachability check for one host to be valid for a different host (and you cannot trust a reachability check for a fake IP address, such as `0.0.0.0`).
- Different IP addresses for the *same* host may be reachable over different interfaces. If a remote host has both IPv4 and IPv6 addresses, iOS typically attempts to connect to both addresses simultaneously, then uses whichever connection was established first and cancels the other connection attempt. If the user's cellular network provides IPv6 and the user's Wi-Fi network doesn't, the connection could be made either using cellular or Wi-Fi, depending entirely on which network connects more quickly.

> **Note:** In iOS 6, the cellular network is never used as the primary interface for IPv4 or IPv6 if Wi-Fi is used as the primary interface for either IPv4 or IPv6, so this particular case is specific to iOS 5 and earlier.

If your app must strictly avoid sending data over a cellular connection, your app must declare that policy restriction explicitly when making the connection. If you are using reachability in an advisory fashion (for example, to warn the user before uploading a large movie over the cellular network), you should also consider making the connection with cellular connectivity disabled. Then, if the connection fails, ask the user for permission to send data over the cellular network and try again without those flags.

At the Foundation layer, you can use the `setAllowsCellularAccess:` method on `NSMutableURLRequest` to specify whether a request can be sent over a cellular connection. You can also use the `allowsCellularAccess` to check the current value.

At the Core Foundation layer, you can achieve the same thing by setting the `kCFStreamPropertyNoCellular` property before opening a stream obtained from the CFSocketStream or CFHTTPStream APIs.

In older versions of iOS, you can continue to use the `kSCNetworkReachabilityFlagsIsWWAN` as a best-effort way of determining whether traffic will be sent over a cellular connection, but you should be aware of its limitations.

Handle Backgrounding Correctly

Your app may be suspended when it goes into the background, which means that it can no longer handle network traffic. In some cases, existing connections may even close while your app is suspended. To learn techniques for coping with backgrounding, read *Networking and Multitasking*.

Register VoIP Sockets Correctly

The `NSInputStream`, `NSOutputStream`, `CFStream`, and `NSURLConnection` APIs have built-in support for Voice over Internet Protocol (VoIP) communication. This support allows you to register a TCP connection as being for VoIP purposes so that if your app gets suspended, data arriving on this socket causes your app to be resumed.

For more information, read Implementing a VoIP Application in *App Programming Guide for iOS*.

Register for Captive Network Support

A captive network is a Wi-Fi network that doesn't provide Internet access until the user performs some action, such as logging in, specifying payment, or agreeing to terms and conditions. Captive networks are common in public areas, such as airports and hotels.

When a user joins a captive network, Captive Network Support typically provides a web sheet that allows the user to authenticate with the network. If your application registers the SSID of the captive network, however, the web sheet is suppressed, and the user can complete authentication in your application.

For more information, read *CaptiveNetwork Reference*.

OS X Lets You Make Systemwide Changes

The following sections explain where to learn about working with network interfaces in OS X and developing network kernel extensions that extend the networking stack.

Develop Network Setup Applications

If you want to modify the current network configuration in your user-level application, use the System Configuration framework.

To learn about the System Configuration architecture, read *System Configuration Programming Guidelines*. Then read *System Configuration Framework Reference* to learn about the available APIs.

If your application specifically deals with connecting to wireless networks with Wi-Fi, you can use the Core WLAN framework. For more information, read *CoreWLAN Framework Reference*.

Develop Network Kernel Extensions

If you want to modify or extend the networking infrastructure of OS X—for purposes such as implementing a custom firewall, a custom VPN, or a bandwidth management system—you may need to write a kernel extension (kext) that plugs in to the kernel's networking subsystem. These extensions are called network kernel extensions, or NKEs.

To learn the fundamentals of writing a kext, read *Kernel Extension Programming Topics*. Then read *Network Kernel Extensions Programming Guide* to learn how to implement a network kext.
