PART I

Understanding iOS and Enterprise Networking

1

Introducing iOS Networking Capabilities

WHAT'S IN THIS CHAPTER?

- Understanding the iOS networking frameworks
- Key networking APIs available to developers
- Using your application's run Loop effectively

Great iOS applications require a simple and intuitive user interface. Likewise, great applications that communicate with a web service of any kind require a well-architected networking layer. An application's architecture must be designed with the flexibility to adapt to changing requirements and the capability to gracefully handle constantly changing network conditions, all while maintaining core design principles that enable proper maintainability and scalability.

When designing a mobile application's architecture you must have a firm grasp of key concepts, such as the run loop, the various networking APIs available, and how those APIs integrate with the run loop to create a responsive, networked application framework. This chapter provides a detailed discussion of run loops and how to use them effectively within an application. Also provided is an overview of the key APIs and when each should be used.

UNDERSTANDING THE NETWORKING FRAMEWORKS

Before you begin development of an iOS application that interacts with the network, you must understand how the networking layers are organized in Objective-C, as shown in Figure 1-1.

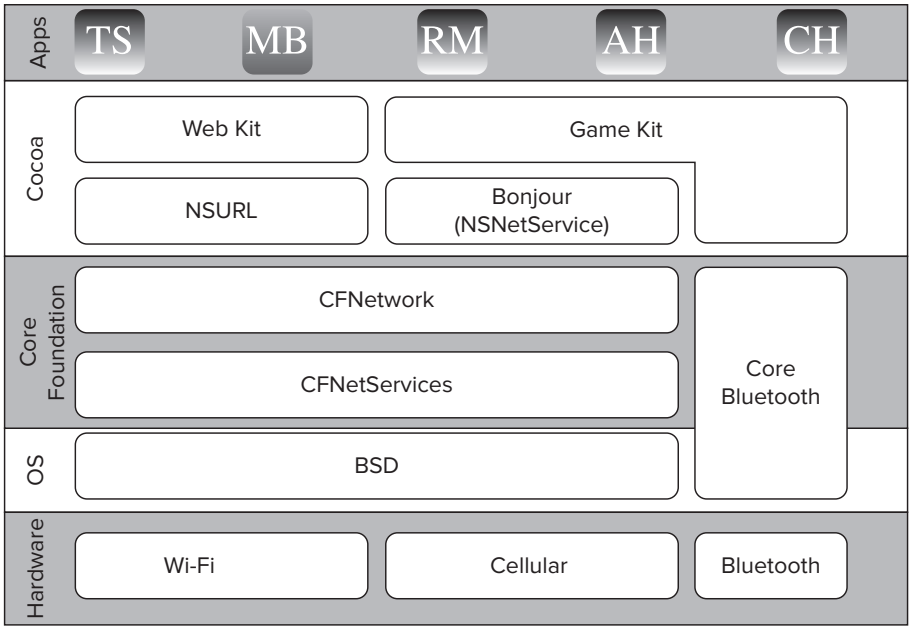

FIGURE 1-1

Each iOS application sits on top of a networking framework stack composed of four levels. At the top is the Cocoa level, which includes the Objective-C APIs for URL loading, Bonjour, and Game Kit. Below Cocoa sits Core Foundation, a set of C APIs that includes CFNetwork, the foundation of most application-level networking code. CFNetwork provides a simple networking interface that sits on top of `CFStream` and `CFSocket`. Those two classes are lightweight wrappers around BSD sockets, which form the lowest level and sit closest to the antenna hardware. BSD sockets are implemented strictly in C and provide developers absolute control over any communication to a remote device or server.

As you move down each level in the framework stack, you tend to gain tighter control but give up the ease of use and abstraction that the previous level provided. Although there are situations in which this may be warranted, Apple recommends that you stay at the CFNetwork layer and above. Raw sockets at the BSD level do not have access to the system wide VPN nor do they activate the Wi-Fi or cellular radios, something CFNetwork handles for you.

Before you design your applications' networking layer you must understand the various APIs available to you and how you can leverage them. The next section covers the key iOS networking frameworks and provides a brief introduction explaining how you can use them. Each API covered is discussed in detail in a future chapter.

iOS NETWORKING APIS

Each level of the framework stack has a set of key APIs that deliver a range of functionality and control to developers. Each level offers more abstraction than the level below it (refer to Figure 1-1). However, this abstraction comes at a cost of losing some control. This section provides an overview of key APIs in iOS and the considerations when using each of them.

NSURLConnection

`NSURLConnection` is a Cocoa level API that provides a simple method to load URL requests, which can interact with a web service, fetch an image or video, or simply retrieve a formatted HTML document. It is built on top of `NSStream` and was designed with optimized support for the four most common URI schemes: `file`, HTTP, HTTPS, and FTP. Although `NSURLConnection` restricts the protocols over which you can communicate, it abstracts much of the lower-level work required to read and write from buffers, includes built-in support for authentication challenges, and offers a robust caching engine.

The `NSURLConnection` interface is sparse, relying heavily on the `NSURLConnectionDelegate` protocol, which enables an application to intervene at many points in the connection life cycle. `NSURLConnection` requests are asynchronous by default; however, there is a convenience method to send synchronous requests. Synchronous requests do block the calling thread, so you must design applications accordingly. Chapter 3, "Making Requests" covers `NSURLConnection` in detail and provides a number of examples.

Game Kit

At its core, Game Kit provides another peer-to-peer networking option to iOS applications. In a traditional network configuration, Game Kit is built on top of Bonjour; however, Game Kit does not require a network infrastructure to function. It can create ad-hoc Bluetooth Personal Area Networks (PAN), which makes it a great candidate for networking in locations with little or no established infrastructure.

Game Kit requires only a session identifier, display name, and connection mode when setting up a network. It does not require configuring of a socket or any other low-level networking to communicate with connected peers. Game Kit communicates via the `GKSessionDelegate` protocol. Chapter 12, "Device-to-Device Communication with Game Kit" discusses integrating Game Kit into your applications.

Bonjour

Bonjour is Apple's implementation of zero configuration networking (zeroconf). Bonjour provides a mechanism to discover and connect with devices or services on the network, and alleviates the need to know a device's network address. Instead, Bonjour refers to services as a tuple of name, service type, and domain. Bonjour abstracts the low-level networking requirements for multicast DNS (mDNS) and DNS-based Service Discovery (DNS-SD).

At the Cocoa level, the `NSNetService` API provides an interface for publishing and resolving address information for a Bonjour service. You can use the `NSNetServiceBrowser` API to discover available services on the network. Publishing a Bonjour service, even with Cocoa level APIs, requires an understanding of Core Foundation to configure sockets for communication. Chapter 13, "Ad-Hoc Networking with Bonjour," includes an in-depth overview of zero configuration networking, Bonjour, and an example of how to implement a Bonjour-based service.

NSStream

`NSStream` is a Cocoa level API built on top of CFNetwork that serves as the foundation for `NSURLConnection` and is intended for lower-level networking tasks. Much like `NSURLConnection`, `NSStream` provides a mechanism to communicate with remote servers or local files. However, you can use `NSStream` to communicate over protocols such as telnet or SMTP that are not supported by `NSURLConnection`.

The additional control that `NSStream` provides does come at a cost. `NSStream` does not have built-in support for handling HTTP/S response status codes or authentication challenges. It transmits and receives data into C buffers, which may be unfamiliar to a strictly Objective-C developer. It also can't manage multiple outbound requests and may require subclassing to add that feature. `NSStream` is asynchronous and communicates updates via the `NSStreamDelegate`. Chapter 8, "Low-Level Networking," and Chapter 13, "Ad-Hoc Networking with Bonjour" cover different implementations of `NSStream`.

CFNetwork

The CFNetwork API is layered on top of the fundamental BSD sockets and is used in the implementations of `NSStream`, the URL loading system, Bonjour, and Game Kit APIs. It provides native support for advanced protocols such as HTTP and FTP. The key difference between CFNetwork and BSD sockets is run loop integration. If your application uses CFNetwork, input and output events are scheduled on the thread's run loop. If input and output events occur on a secondary thread, it is your responsibility to start the run loop in the appropriate mode. The "Run Loops" section later in this chapter provides additional details.

CFNetwork provides more configuration options than the URL loading system, which can be both beneficial and frustrating. These configuration options are visible when creating an HTTP request with CFNetwork. When creating the request you must manually add any HTTP headers and cookies that must be transmitted with the request. With `NSURLConnection`, though, standard headers and any cookies in the cookie jar are automatically added for you.

The CFNetwork infrastructure is built on top of the CFSocket and CFStream APIs from the Core Foundation layer. CFNetwork includes APIs for specific protocols such as CFFTP for communicating with FTP servers, CFHTTP for sending and receiving HTTP messages, and CFNetServices for publishing and browsing Bonjour services. Chapter 8 covers CFNetwork in greater detail, and Chapter 13 provides an overview of Bonjour.

BSD Sockets

BSD sockets form the basis for most Internet activity and are the lowest level in the networking framework hierarchy. BSD sockets are implemented in C but can be used within Objective-C code. Use of the BSD socket API is not recommended because it does not have any hooks into the operating system. For example, BSD sockets are not tunneled through the system wide VPN nor do any of the API calls automatically activate the Wi-Fi or cellular radios if they are powered down. Apple recommends that you work solely with at least CFNetwork or higher. Chapter 8 covers BSD sockets and CFNetwork in greater detail and provides examples of how they can be integrated into your application.

As you implement the various network APIs, you must understand how they integrate with your application. The next section discusses the concept of run loops, which monitor for network events (among other things) from the operating system and relay those events to your application.

RUN LOOPS

Run loops, represented by the class `NSRunLoop`, are a fundamental component of threads that enable the operating system to wake sleeping threads to manage incoming events. A run loop is a loop configured to schedule tasks and process incoming events for a period of time. Each thread in an iOS application can have at most one run loop. For the main thread the run loop is started for you and is accessible after the application delegate's `applicationDidFinishLaunchingWithOptions:` method is invoked.

Secondary threads, however, must run their run loop explicitly, if needed. Before starting a run loop in a secondary thread, you must add at least one input source or timer; otherwise, the run loop exits immediately. Run loops provide developers with the ability to interact with a thread, but are not always necessary. Threads spawned to process a large data set without any other interaction, for example, probably do not warrant starting the run loop. However, if the secondary thread interacts with the network, you need to start the run loop.

There are two source types from which run loops receive events: input sources and timers. Input sources, which are typically either port-based or custom, deliver events to the application asynchronously. The primary difference between the two types of sources is that the kernel signals port-based sources automatically, whereas custom sources must be signaled manually from a different thread. You can create a custom input source by implementing several callback functions associated with `CFRunLoopSourceRef`.

Timers generate time-based notifications that provide a mechanism for applications (threads specifically) to perform a specific task at a future time. *Timer events* are delivered synchronously and are associated with a specific mode, which is discussed later in this section. If that particular mode is not currently monitored, events will be ignored, and the thread will not be notified until the run loop is "run" in the corresponding mode.

You can configure timers to fire once or repeatedly. Rescheduling is based on the scheduled fire time, not the actual fire time. If a timer fires while the run loop is executing an application handler method, it waits until the next pass through the run loop to call the timer handler, typically set via `@selector()`. If firing the handler is delayed to the point in which the next invocation occurs, the timer fires only one event with the delayed event being suppressed.

Run loops can also have observers, which are not monitored and provide a way for objects to receive callbacks as certain activities in the run loop execution occur. These activities include when the run loop is entered or exited, as the run loop goes to sleep or wakes up, and before the run loop processes an input source or timer. They are documented in the `CFRunLoopActivity` enumeration. Observers can be configured to fire once, which removes the observer after it has been fired, or repeatedly. To add a run loop observer, use the Core Foundation function `CFRunLoopObserverRef()`.

Run Loop Modes

Each pass through the run loop is run in a specific mode specified by you. *Run loop modes* are a convention used by the operating system to filter the sources that are monitored and allowed to deliver events, such as calling a delegate method. Modes include the input sources and timers that should be monitored as well as any observers that should be notified of run loop events.

There are two predefined run loop modes in iOS. `NSDefaultRunLoopMode` (`kCFRunLoopDefaultMode` in Core Foundation) is the system default and should typically be used when starting run loops and configuring input sources. `NSRunLoopCommonModes` (`kCFRunLoopCommonModes` in Core Foundation) is a collection of modes that is configurable. Assigning `NSRunLoopCommonModes` to an input source by calling a method such as `scheduleInRunLoop:forMode:` on an input source instance associates it with all modes currently in the group.

> **NOTE** *OSX includes three additional predefined run loop modes that you may see referenced in different documentation.* `NSConnectionReplyMode`*,* `NSModalPanelRunLoopMode`*, and* `NSEventTrackingRunLoopMode` *provide additional filtering options but are not available on iOS.*

Although `NSRunLoopCommonModes` is configurable, it is a low-level process that requires calling the Core Foundation function `CFRunLoopAddCommonMode()`. This automatically registers input sources, timers, and observers with the new mode instead of manually adding them to each new mode. You can define custom run loop modes by specifying a custom string such as `@"CustomRunLoopMode"`. For your custom run loop to be effective, you must add at least one input source, timer, or observer.

Although this provides an overview of run loops, Apple provides several in-depth resources on run loop management that you should review if you develop advanced, network-based, and multi-threaded applications. The developer documentation is available at `https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html`. Networking techniques that benefit from run loop integration are discussed in their respective chapters such as Chapter 8, “Low-Level Networking” and Chapter 13, “Ad-Hoc Networking with Bonjour.”

SUMMARY

Understanding the iOS networking stack and how applications interact with the run loop is an important tool in the iOS developer’s belt. A well-architected networking layer provides incredible flexibility to an application. Likewise, a poorly designed networking layer can be detrimental to its success and ability to scale.

The tools presented in this chapter provide an overview of the various networking APIs and how they compare. How they are applied, although covered briefly here, is discussed in detail in the upcoming chapters.