Chapter 6

Networking: Connections, Data, and the Cloud

In today's world of ubiquitous mobile devices and always-on connections to the Internet, the topic of networking is of great importance. On iOS alone, a great quantity of applications make use of the network in one way or the other, whether using built-in systems like iCloud synchronization or GameKit's communication framework, or by communicating directly with servers over the Internet. Moreover, the expectations of consumers often clashes with the capabilities and limitations of wide-area mobile networks such as EDGE, 3G, and HDPSA+, leaving you, the application programmer, to make the best of a high-latency, low-throughput network channel.

Well, luckily for you, I have a positive ocean of information and experience to share on this front. In this chapter you'll learn about the URL loading system, and asynchronous, synchronous, and synthetic-synchronous network access. You'll learn the best ways to handle common types of data passed across the network and how to correctly handle binary data in a system-agnostic fashion. You'll also see how to locate services on the network plus how you can create your own and make their information available in the same manner.

Perhaps most importantly, though, you'll learn the rationale behind the following core tenets of network-centric programming on the Mac and iOS:

- Never access the network from the main thread.
- Use asynchronous network primitives wherever possible.
- Use streams for all data handling.
- Keep data handling and transport handling unaware of one another.

Stick to these four rules and you'll not go far wrong. In fact, you'll be in a much better position to handle anything the network can throw at you—because it will throw everything it can, sooner or later.

J. Dovey et al., *Beginning Objective-C*

Basic Principles

You've already learned a lot about the local filesystem (yes, you're allowed to exclaim something about "understatement" at this point), and while the data access APIs on the Mac and iOS are frequently storage-agnostic, there can be a world of difference between loading a file from disk and loading it from the network. To underline the lessons of this chapter, I'm going to take a few minutes to explain these differences and give you a good grounding point for anything you might learn about networking in future.

With both network and local-file access there are a certain set of problems that might impact your application's performance. These fall into two broad groups:

- *Latency* refers to the amount of time it takes to reach the resource you want to access. In terms of local disks, this usually refers to the seek-time of the underlying storage medium. In network terms, this refers to the round-trip time of a command packet.
- *Throughput* refers to the speed at which data can be sent to or retrieved from the resource. Local disks usually have very high throughput and are able to move hundreds of megabytes in a second. On a network this could be anywhere from tens of megabytes per second to a few kilobytes per second.

Also associated with these two types is the available bandwidth. For local storage, the reads and writes typically travel through a hardware bus, which has much higher throughput than the hard disks themselves. So unless a lot of disks are being used at the same time, the overall throughput is likely to match that of the physical media in use by a particular operation. This means that if you read or write to multiple devices at the same time you will likely not see any appreciable change in the throughput from each device compared to handling that device alone.

On the network, however, the available bandwidth is usually much more restrictive, and each single operation can easily saturate the connection, eating up all the available bandwidth. As a result, performing more operations in parallel will likely result in all the operations slowing down as the available bandwidth is apportioned between them. In addition, there are more nodes on the journey between your device and the actual resource, such as wireless access points and routers. Any of these might have their own bandwidth concerns and any number of other clients, and they might have specific rules that they enforce to prevent any single client from using up too much bandwidth. For an example, many common access points and routers will allow only four concurrent requests using the HTTP protocol to go from a single client to a single destination resource; all others are blocked until some of the first four are done. As a result, it is often useful for HTTP client software to implement the same limit internally to avoid having many concurrent operations spinning their wheels and using up local resources like memory, CPU, and battery power.

On the flip side, there is frequently little difference in requesting a number of different resources from the same location via the network, while doing so with files on the same local disk can cause a lot of slow-down. The reason is down to the cause of latency on disk-based storage: the seek time. When a single large file is being read, the disk will seek to the file once and then read a (largely) contiguous stream of data. When multiple

files are being read and written in parallel, the disk's read head needs to move back and forth across the magnetic platter to switch between the files frequently, leading to that phenomenon known as *disk thrashing*—the clunking and clicking sounds that you can hear coming from your computer during periods of high activity. Recent advances in solid-state storage mean that a lot of people now use solid state drives (SSDs) rather than platter-based magnetic hard disk drives, so the latency issue is much less of a problem on systems equipped with such storage. The iPhone, of course, uses solid-state storage.

Network Latency

Latency on network transmissions is most frequently caused by either congestion of the network (which can occur at any stop on the journey, remember) or by inefficiency of the underlying transmission protocol. Wired network connections and Wi-Fi wireless connections use versions of the Ethernet protocol. This protocol is responsible for locating other devices on the local network and ferrying data between them in an ordered fashion. Mobile devices using the cellular network, however, use a protocol such as GSM or HSDPA, which is working in a much more error-prone environment and needs to maintain data throughput even when devices are moving at high speed between endpoints such as cell towers. As a result, those protocols have significant overhead when compared to Ethernet. This isn't helped when you consider that most Ethernet or Wi-Fi devices have throughputs in the tens or hundreds of megabits per second while cellular networks are only just beginning to reach past the 15 Mbit boundary with HSDPA+ and "4G"[1]. As a result, it is common to see a latency of up to a full second (even two!) when making a connection over a cellular network, particularly over the slower protocols like GSM or 3G.

If your application is making a lot of connections, for instance when fetching data from a server over the HTTP protocol, then latency can really cause you problems. Imagine you need to make about 20 separate calls to pull down a full list of 200 comments (the server returns 10 per page). Every call will by default make a new connection, and each connection will encounter a long delay due to latency. If the latency is quite bad, then regardless of the actual throughput speed you're guaranteed to see this take somewhere between 10 and 40 seconds, which isn't a very good experience. If the throughput is also slow, it becomes even worse. I've seen an implementation of syncing data from such an API take over an *hour* when there are a lot of calls to be made, and almost half of that time was due to network latency.

Asynchronicity

With all these potential pitfalls waiting to trip you up, it becomes *vitally important* that you interact with network resources in a fully asynchronous manner. I can't state this strongly enough: it *must always be asynchronous, no matter what*.

[1]I put this in quotes because the 4G standard actually specifies that there should be one *gigabit* per second of bandwidth to attain that label.

It's easy to fall into the trap of convincing yourself that something doesn't need to be done in an asynchronous manner. Perhaps you're only using the network occasionally; perhaps you're only sending or receiving very small amounts of data. Perhaps you only send data, but don't wait for a reply, or you only perform network operations in response to user input. We've heard all these excuses, and I have some news for you: *the network will still get you*. For example:

- *I only use the network occasionally.*
  If you're doing it occasionally, the user likely isn't expecting it. If your occasional activity causes the application to become unresponsive, your user is likely to be more upset than if it were a regular occurrence, because they're unable to predict when it will happen.
- *I only send very small packets of data.*
  Small packets might move quite quickly even on networks with low throughput, but network latency doesn't care about the packet size. The action of making the connection could take long enough that the application becomes unresponsive, even for a short time, and that's a Bad Thing.
- *I only send; I never wait for a reply.*
  You might not be waiting for a reply, but your send operation won't complete until the network stack receives a confirmation that the data was received correctly. You could use a protocol with non-guaranteed transfer of data, but for sending user data somewhere that's a bad idea—you *want* to know the data got there safely.
- *It's always caused by user input, so they're conditioned to expect some small delay.*
  The user might change their mind. If you're doing everything synchronously, you don't have any way to asynchronously cancel the operation if the user decides it's taking too long.

If you have another reason to stick with synchronous networking, please let me know what it is—I'll be only too happy to shoot it down.

Sockets, Ports, Streams, and Datagrams

The networking system on OS X and iOS uses the BSD sockets system. This essentially makes a network connection appear similar to a file: data is sent and received by writing to or reading from a standard file handle, and the kernel takes care of the rest. There are some extra capabilities, but for your purposes you're unlikely to need to look at them here. The main difference when dealing with network connections is that data isn't available on-demand, nor is the space in which to write. Incoming data is available only once it arrives through the transport medium, and outgoing data is stored temporarily in a system buffer until the system is able to send it across the wire. When you looked at `NSStream` in the previous chapter, you saw events and methods based around these concepts: data available to read and space available to write.

A network connection's file handle is called a *socket*. Once created, these can be used almost like regular file handles, right up to the point of wrapping it with an `NSFileHandle` instance. The chief difference is the inability to seek to a location in the data stream—it's strictly an ordered-access system.

There are two primary types of sockets:

- *Stream sockets* are designed to send large collections of ordered data from one end of a connection to the other. The underlying transport typically provides guaranteed delivery of content—and in the correct order.
- *Datagram sockets*, on the other hand, do not provide any guarantees of either delivery or ordering. Each parcel of data is considered a separate stand-alone element, and no attempt is made to ensure that the datagrams follow the same route through the network. Each datagram could follow a different route, depending on the conditions at each node on the journey.

Datagram sockets, normally using the Unified Datagram Protocol (UDP) over the Internet Protocol (IP), are typically used for things like streaming media, where a datagram from a certain location in the underlying data stream is expected to arrive at a certain time, but the data is of a type that can be reassembled reasonably well without receiving every byte in order. For example, if you've ever watched a live webcast you've probably seen the speaker's image replaced at some point by various pieces of some earlier frame in the video, seemingly warped and wrapped to match the speaker's movements. This is an example of missing datagrams. The video data provides a full frame only every now and then, and in between these the data consists only of modifications to apply to that frame to shape it into the current frame. When a full frame is missed, this results in the strange effect of warping the wrong frame image to create a new one, causing the visual effect described above.

Most network communication, however, uses streaming sockets, typically using the Transmission Control Protocol (TCP) over IP. TCP provides guaranteed delivery at the expense of guaranteed time-to-destination. If something goes wrong partway through the transfer, the receiving endpoint is able to check that it received the packet correctly and request a resend if it didn't arrive properly. The TCP protocol also ensures that ordering is maintained, even when some packets arrive correctly and some do not; if packet A fails and packet B succeeds, the contents of packet B will be kept around until the content of A can be delivered successfully.

TCP also implements *transmission windows*. These allow each endpoint to communicate to the other some details on the amount of data it can successfully handle. As a result, unlike UDP (Unified Datagram Protocol), one endpoint never sends any data that the receiver cannot currently handle. Datagram protocols, on the other hand, are designed to pump data out as fast as possible (or at least at a continuous rate), allowing the receiver to just skip any packets that arrive before it can handle them.

In all the examples that follow, I will be using TCP/IP, rather than datagrams. Datagram protocols typically require more work at the application level to correctly handle errors such as missing packets, which is *very definitely* beyond the scope of this book.

Underneath the TCP and UDP protocols lies the Internet Protocol. This is the source of many things you may recognize already: IP addresses as dotted octets such as 192.168.23.5, and stem from version 4 of the IP protocol. An IP address references a single host on the network; each connection endpoint is made available through a *port*

on that address, which is a 16-bit number identifying a particular endpoint. There are a number of standard port assignments registered with the relevant authorities, so certain services are usually accessible through a known port number. HTTP, for example, uses port 80, while secure HTTP uses port 443. The Apple Filing Protocol, when served over TCP/IP, uses port 548, and there are many, many more standard assignments in use—a fair chunk of the available 65535 ports supported by the protocol.

> **Note** The term "dotted octet" refers to the textual representation of what is actually just a 32-bit number. Each of the four segments of an IP address represents a single byte, and the word "octet" is another word for "byte"—a byte contains eight bits. Once upon a time different systems used different sizes of bytes, hence the remaining use of the word "octet" here.

The 32-bit addressing scheme used by IPv4 (read: Internet Protocol, version four) only allows for around 4.3 billion unique addresses. Add to that the fact that large ranges of these addresses are off-limits for general use—for use only as local-network private addresses, or for different types of communication, such as multicast IP—then it's hardly surprising to discover that we've actually run out. Service providers snapped up the last available IPv4 addresses in early 2012. The solution to this is the current rollout of IPv6, the latest version of the standard. This update uses a 128-bit addressing system, which is expected to last far beyond the lifetimes of anyone alive today. From the software's point of view, however, the only difference is in how an address is specified. As you'll see later in this chapter, this is frequently obviated by service discovery systems, which remove the need for handling addresses directly. For those occasions when you do need to work with them, the same POSIX routines that work with IPv4 addresses also handle IPv6 out of the box.

The Cocoa URL Loading System

The highest-level API for accessing network resources on OS X and iOS revolves around the judicious use of `NSURL`s. There is a large framework behind these objects, which is invoked either automatically by object methods such as `-initWithContentsOfURL:` or explicitly through the `NSURLRequest` or `NSURLConnection` classes. A graphical overview of the URL system can be seen in Figure 6-1.

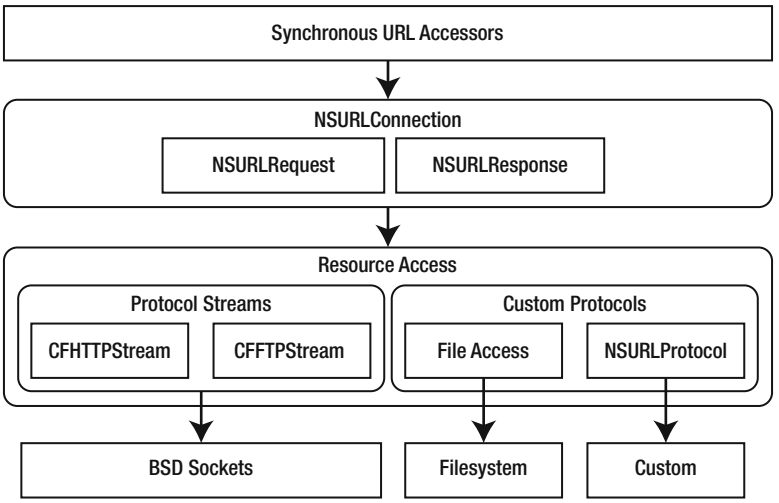

***Figure 6-1.** Components of the Cocoa URL loading system*

The URL loading system itself provides out-of-the-box support for a number of commonly used protocols:

- File Transfer Protocol (FTP), using `ftp://server.com/path/to/item`.
- Hypertext Transfer Protocol (HTTP), using `http://server.com/path/to/item`.
- Secure Hypertext Transfer Protocol (HTTPS), using `https://server.com/path/to/item`.
- Local files, using `file:///path/to/item` or `file://localhost/path/to/item`.

It also provides the means for you to implement your own URL-based protocol handling via the `NSURLProtocol` class. This is beyond the scope of this book, but it's worth looking at the headers and documentation if you'd like to provide URL-based access to the interior of a container file such as a zip or xar archive or implement URL support for your own custom protocol.

The URL loading system also has support for handling authentication on any of the above protocols using a challenge-response format. Making use of this requires you to use the `NSURLConnection` class directly to manage your resource access, however, as you'll need to provide a delegate object to respond to any authentication challenges which arise. You will see this later.

All URL loads are performed using asynchronous methods. Whether created by you or created by a system framework on your behalf, an `NSURLConnection` lies at the heart of all URL access operations. The difference is that when using one-shot synchronous methods to load data from a URL, you are in fact using a *synthetic synchronous* method. This is because the method itself is using an asynchronous API based on callbacks and delegation to implement the data

access, but is silently waiting for it to complete before returning. This is better than a plain synchronous approach (at least this way the waiting is done in a manner that allows other processing to occur) but still not as good as programming directly against the asynchronous API. For instance, if a synthetic synchronous method waits by spinning the current run-loop in the default mode, then it would be bad for it to do so while the current run-loop is actually in an event-tracking mode. And if it simply ran the current run-loop in its current mode, the same thing might happen if the event-tracking sequence ends: this thing is still running a non-default run-loop while it waits for an indeterminate amount of time.

The actual transfer of data is typically implemented using streams. The core implementations of URL connections, requests, and responses (in `CFNetwork`, a CoreFoundation-based framework) have custom stream classes to perform the work of communicating using a particular protocol. Your `NSURLProtocol`-based accessors may or may not use streams internally, though the `NSStream` APIs do provide perhaps the most cooperative set of methods to implement a properly asynchronous protocol handler. Note that the stream classes used to implement the built-in protocol support are usually entirely private, conforming only to the public interfaces defined by `NSInputStream` or `NSOutputStream`. In fact, if you're using `NSURLConnection` you'll typically never see the streams themselves as the connection object hides the underlying objects and hands its delegate already-accumulated data packages and high-level events.

Using NSURLConnection

Loading data from a remote resource (or sending data to it) begins with a request, encapsulated by the `NSURLRequest` object. At its simplest, a URL request contains a URL, a timeout (the default is 60 seconds), and a specification of some caching behavior. Each protocol defines its own default caching behavior, but you can choose to override this by requesting that only cached data is returned, or that the cache should be ignored and the remote resource always refetched. There is also an `NSMutableURLRequest` class that allows you to modify any of these attributes and supply some additional metadata, such as an associated base document URL or whether to allow the request to utilize an available cellular network.

For HTTP requests, there are some additional methods available to set some of the attributes of the request, specific to the HTTP protocol. These allow you to set the method and any header values, as well as supply body data for the request either as an `NSData` object or an `NSInputStream` instance. It is usually *much* better to provide an input stream if at all possible, especially if you're doing something like uploading a file. To do so, you could just create a new input stream using the local file's URL and provide that to the `NSURLRequest` object. You can also tell the URL request to attempt to use HTTP pipelining to send multiple requests in sequence without waiting for replies for each one. Note that whether this actually happens is up to the server, which may not support pipelining.

Listing 6-1 shows an example of creating a URL request to fetch the contents of Apple's home page.

Listing 6-1. Fetching Apple's Home Page using a HTTP Request

```
NSURL *url = [NSURL URLWithString: @"http://www.apple.com/"];
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL: url];
[request setHTTPMethod: @"GET"];
[request setHTTPShouldUsePipelining: YES];
```

Once you have your request, you use it to initialize an `NSURLConnection` instance. This object will then take on the handling of the actual transmission of the request and the receipt and parsing of its response. By default a new `NSURLConnection` instance will schedule itself with the current run-loop's default mode and immediately send its request and begin processing, although you can override this behavior if you wish and use the `-start` method to kick off the process, as seen in Listing 6-2.

Listing 6-2. Initializing an NSURLConnection

```
NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:request
                                    delegate:delegate startImmediately:NO];

// you can either schedule the connection on a run loop or an operation queue
if ( [self shouldUseRunLoop] )
    [connection scheduleInRunLoop:[NSRunLoop currentRunLoop] mode:NSDefaultRunLoopMode];
else
    [connection setDelegateQueue: [self operationQueue]];

// fire up the connection, sending the request
[connection start];
```

NSURLConnection uses a delegation system to handle the events that occur, providing the delegate with any returned data (for HTTP, this refers to *body* data; the headers are parsed into a response object automatically), reporting any errors, and requesting confirmation or input for any authentication challenges or redirection it might encounter.

The delegate methods for NSURLConnection are split into three separate protocols:

- `NSURLConnectionDelegate` declares methods that allow the delegate to handle errors and authentication challenges.
- `NSURLConnectionDataDelegate` adds methods that notify the delegate of the receipt of responses and data, and allow the delegate to modify outgoing requests due to redirection and to monitor the transfer of sent data. In the case where a stream was provided for body data, the delegate will also be asked to provide a new stream should the request need to be restarted.
- `NSURLConnectionDownloadDelegate` adds methods that notify the delegate of the process of downloading a file to local storage. This functionality is only available on iOS within Newsstand applications, however, so I will not cover it in this chapter.

Both `NSURLConnectionDataDelegate` and `NSURLConnectionDownloadDelegate` incorporate the methods declared by the `NSURLConnectionDelegate` protocol. You can think of them as being members of a class hierarchy.

Let's break down the delegate methods into groups. First you'll look at authentication, which is implemented using the `NSURLAuthenticationChallenge` and `NSURLCredential` classes.

Authentication

The URL connection notifies its delegate of an authentication challenge using two methods. Firstly it will check whether it ought to use the built-in credential store to look up authentication information automatically by calling `-connectionShouldUseCredentialStorage:` when the connection is started. If the delegate does not implement this method, the connection will assume that the answer is `YES`.

When an authentication challenge is received from the remote service, this value will be used to construct a default challenge response. If the credential store was consulted, this response may already contain valid credentials. Either way, if implemented the delegate's `-connection:willSendRequestForAuthenticationChallenge:` method, providing details on the challenge and any applicable saved credential that the system would attempt to use to meet the challenge. Usually you will want to implement this method because doing so is likely the primary means by which credentials will be added to the store for later use.

The second parameter is an instance of `NSURLAuthenticationChallenge`, which encapsulates all available information about the challenge itself. Through this object you can determine the authentication method, the proposed credential (if one was available from the store, or if a protocol implements a common "anonymous" credential), the number of previous failed attempts, and more.

For example, you can access an `NSURLProtectionSpace` object that describes the domain against which the credential would be applied; this normally refers to the host, port, and protocol used, and may (depending on the protocol) specify a realm to further narrow down the authentication's applied scope. For example, a standard HTTP connection might specify a protocol of `HTTP`, a host of `www.apple.com`, and a port of `80` (the default port for HTTP services). An example realm for HTTP might be a subfolder: perhaps most of the host is accessible, but getting to `www.apple.com/secure/` requires a credential, so the realm would be "`secure`" since the credential would only provide access to items within that folder resource. The protection space also provides information on whether the credential will be transmitted securely, along with the details of things like the certificate trust chain for a secure connection. You can use any of this to make a decision on whether to return a valid credential back to the server.

The core of the matter revolves around the creation of an `NSURLCredential` object. In order to authenticate against a service you need to create a credential and provide that to the sender of the authentication challenge, accessed via the authentication challenge's `-sender` method. The authentication challenge sender itself is any object that corresponds to the `NSAuthenticationChallengeSender` protocol, which declares the methods shown in Table 6-1. You use those methods to inform the challenge sender of your decision regarding the current authentication attempt: you can provide a credential, cancel the authentication, and more.

Table 6-1. Authentication Challenge Sender Methods

Method	Description
`- (void)useCredential:(NSURLCredential *) credential forAuthenticationChallenge: (NSURLAuthenticationChallenge *)challenge`	Call this method to provide a credential used to attempt authentication.
`- (void)continueWithoutCredentialFor AuthenticationChallenge: (NSURLAuthenticationChallenge *)challenge`	Attempts to continue access without providing a credential. Typically this will result in some alternative result; for instance an HTTP server might return a HTML page containing a descriptive error and further instructions.
`- (void)cancelAuthenticationChallenge: (NSURLAuthenticationChallenge *)challenge`	Cancels the authentication attempt, which typically causes the connection to abort with an error.
`- (void) performDefaultHandlingForAuthenticationChallenge: (NSURLAuthenticationChallenge *)challenge`	Instructs the sender to fall back on default behavior, as if the URL connection's delegate did not implement the authentication delegation methods. Optional.
`- (void)rejectProtectionSpaceAndContinueWithChallenge: (NSURLAuthenticationChallenge *)challenge`	Rejects the authentication protection space and continues trying to access. In some protocols this may result in a different form of authentication being attempted. Optional.

It's worth noting that by declaring this interface as an Objective-C protocol, it is possible to create authentication challenges for your own custom URL protocol handlers. The `NSURLAuthenticationChallenge` method takes a reference to a sender object in its constructor, and (unlike some classes in the URL loading system[2]) its creation API is publicly available to use.

You most likely implementation, however, is going to involve simply asking the user to provide logon credentials at some point. You could then keep these in the credential store or place them manually in the user's keychain for later secure access. The latter is beyond the scope of this book, so for an example of proper handling of the former look no further than Listing 6-3. Note that it is *absolutely required* that you call one of the methods from Table 6-1 in response to a `-connection:willSendAuthenticationChallenge:` message, as the connection attempt will not proceed until you have done so.

[2]NSHTTPURLResponse, we're looking at you. Yes, you with the initializer that was only introduced in iOS 5 and OS X 10.7, despite the class existing all the way back to OS X 10.0.

Listing 6-3. Implementing Authentication in an NSURLConnectionDelegate

```
- (BOOL) connectionShouldUseCredentialStorage:(NSURLConnection*)connection
{
    return ( YES );
}

- (void) connection:(NSURLConnection*)connection
        willSendAuthenticationChallenge:(NSURLAuthenticationChallenge*)challenge
{
    // look at the challenge-- does it have a proposed credential?
    if ( [challenge proposedCredential] != nil )
    {
        // has it failed already? If not, try it
        if ( [challenge previousFailureCount] == 0 )
        {
            // use the proposed credential and return
            [[challenge sender] useCredential: [challenge proposedCredential]
                   forAuthenticationChallenge: challenge];
            return;
        }
    }

    // if we get here, then either there's no proposed credential or it failed
    // that means we need to ask for one
    NSString *user = nil, *pass = nil;
    if ( [self promptForAuthenticationForChallenge:challenge
                                    returningUser: &user pass: &pass] == NO )
    {
        // the user cancelled the dialog, so cancel authentication
        [[challenge sender] cancelAuthenticationChallenge: challenge];
        return;
    }

    // if there was no username, try the default handling
    if ( [user length] == 0 )
    {
        // this method is optional, so check for it
        if ( [[challenge sender] respondsToSelector:
                        @selector(performDefaultHandlingForAuthenticationChallenge:)] )
        {
            [[challenge sender] performDefaultHandlingForAuthenticationChallenge:
                                                                        challenge];
        }
        else
        {
            [[challenge sender] continueWithoutCredentialForAuthenticationChallenge:
                                                                        challenge];
        }

        // we've made a decision, so return now
        return;
    }
```

```
    // Note that it's perfectly valid to send an empty password. It's up to the server
    // to decide whether that's allowed upon receipt.
    NSURLCredential * credential = [NSURLCredential credentialWithUser:user
                                        password:pass
                                        persistence:NSURLCredentialPersistenceForSession];
    [[challenge sender] useCredential: credential
           forAuthenticationChallenge: challenge];
}
```

In cases where authentication is based upon server-side and client-side certificates, it is possible to initialize an `NSURLCredential` instance with a certificate chain and identity or by indicating that you have chosen to trust a secure identity provided by the server. In some cases, both of these may be required in separate steps: first you choose whether to trust the server, then you provide a secure identity so the server can decide if it trusts you.

URL Connection Data Handling

When you implement an object confirming to the `NSURLConnectionDataDelegate` protocol, you can monitor or affect four areas of the transaction:

- You can allow, deny, or modify any requests sent as a result of receiving a redirection request (only appropriate for certain protocols).
- You can observe and store the parsed URL response (an instance of `NSURLResponse` or its subclass `NSHTTPURLResponse`), handle packets of data as they arrive, and be notified when the connection has completed all its work.
- You can monitor the flow of data sent in the body of a request and provide a new copy if required to resend.
- You can affect the storage of a URL response in the URL system's response cache by allowing or denying it or by modifying or replacing a proposed `NSCachedURLResponse` object.

All of these methods are optional. In most cases, you'll likely only implement `-connection:didReceiveData:` and `-connectionDidFinishLoading:`. Among the common reasons for implementing the confirmation methods, however, is to implement some basic security checking. For example, Listing 6-4 shows implementations of the redirection handler and caching handler, which think about security considerations.

Listing 6-4. Security Considerations in NSURLConnectionDataDelegate

```
- (NSURLRequest*)connection:(NSURLConnection*)connection
            willSendRequest:(NSURLRequest*)request
           redirectResponse:(NSURLResponse*)response
{
    // we know the server we're talking to, and it will never send us elsewhere
    // therefore, we do NOT accept redirects to a different host
    NSString *responseHost = [[response URL] host];
    NSString *newRequestHost = [[request URL] host];
    if ( [responseHost isEqualToString: newRequestHost] == NO )
```

```
    {
        // this is unexpected behaviour from a secure server: don't follow
        // return nil to reject the request
        return ( nil );
    }

    // we're going to allow the redirect
    // if we wanted to ensure certain headers are set (i.e. a content hash)
    //  we would create a mutable copy of the request and modify and return it

    // for now, let's just return the proposed request to use that, unchanged
    return ( request );
}

- (NSCachedURLResponse*)connection:(NSURLConnection*)connection
                willCacheResponse:(NSCachedURLResponse*)cachedResponse
{
    // don't cache anything from within our secure realm
    // our secure realm is inside '/secure' on the server, so any URLs whose
    //  paths begin with that component should not be cached, ever
    NSArray *components = [[[cachedResponse response] URL] pathComponents];
    if ( [components count] > 0 &&
         [[components objectAtIndex: 0] isEqualToString: @"secure"] )
    {
        // the response is from the secure domain, so don't cache it
        return ( nil );
    }

    // non-sensitive data can be cached as-is
    return ( cachedResponse );
}
```

Handling responses and data is as simple as implementing the appropriate methods and inspecting the values passed in: `-connection:didReceiveResponse:` and `-connection:didReceiveData:` respectively. Handling of both of these types is simple enough that I won't provide explicit examples here (in 99% of cases, the response and data are just put into local variables), but I would like to make one rather important suggestion: *don't accumulate response data in memory*.

Well, ok, small amounts are safe enough, but if you're talking to an API that might return kilobytes of data, it is *much* better to drop it all into a temporary file somewhere. Once all the data has been downloaded you can either read it from the file in a stream or you can use memory-mapping (via `NSData`'s `-initWithContentsOfURL:options:error:`) to get a regular data object without allocating potentially large amounts of memory. This is more important for iOS, of course, but it's a good idea and a good habit to form. Listing 6-5 shows how you might handle this.

Listing 6-5. Accumulating Returned Data in a Temporary File

```
- (id)init
{
    self = [super init];
    ...

    self.tmpFilePath = [NSString pathWithComponents: @[NSTemporaryDirectory(), ...]];
    self.tmpFile = [NSFileHandle fileHandleForWritingAtPath: path];
    ...

    return ( self );
}

- (void)dealloc
{
    // delete the temporary file, we don't need it any more
    [[NSFileManager defaultManager] removeItemAtPath:self.tmpFilePath error:NULL];
}

...

- (void)connection:(NSURLConnection*)connection didReceiveData:(NSData*)data
{
    // push it out to the temp file
    [self.tmpFile writeData: data];
}

...

- (void) connectionDidFinishLoading:(NSURLConnection*)connection
{
    // close the file, we don't need to access it any more
    [self.tmpFile closeFile];

    // get a cheap NSData object by memory-mapping the file
    // we also specify uncached reads to avoid building up kernel
    // memory buffers for data we'll only read once
    self.data = [NSData dataWithContentsOfFile:self.tmpFilePath
                                options:NSDataReadingMappedAlways|NSDataReadingUncached
                                error:NULL];
}
```

Using this approach you can successfully receive and handle a large amount of data while keeping your application's memory footprint low, which in this time of ubiquitous mobile computing is a very useful trait indeed.

Network Streams

You might be thinking to yourself, "Why would I accrue all this data immediately? What about the stream APIs I worked with in Chapter 5? Didn't you say they were especially useful for network programming?"

Well, you're absolutely right. You've already seen that an NSURLRequest can take an NSInputStream from which to read any body data associated with the request. What you haven't seen yet is how to provide the incoming data as a stream; the data only arrives through the NSURLConnectionDataDelegate methods, which makes them of lesser utility when compared to pure streams. After all, none of the standard system classes have methods that take arbitrary partial data packages, but a number do have stream support:

- NSPropertyListSerialization supports reading data from a stream to create property list types (dictionaries or arrays containing strings, numbers, dates or raw data) and writing serialized data to an output stream.
- NSXMLParser can parse data received from an input stream as well as the contents of a URL or a provided data blob.
- NSJSONSerialization will both read from and write to stream classes to parse and create JSON (JavaScript Object Notation) data.

I'm sure you noticed the common theme amongst the three classes listed above: they all deal with serializing or parsing structured data, and two of them deal with types that are widely used for sending data across networks. Property lists might sound like an odd choice, but at WWDC 2010, David den Boer (hi Dave!), manager of the team that created the Apple Store app for the iPhone, revealed that by using property lists for data transfer they saw a *massive* improvement in throughput and thus, by extension, application responsiveness[3].

Now, let's say you've requested some XML data from a HTTP server, and the data comes back in 800 16KB blocks passed to your -connection:didReceiveData: method. That's about 12MB of XML. That much data could take a while to download, depending on the network conditions at the time. If you download it to a file like I suggested, you're going to spend a fair amount of time fetching the data before you've even begun to think about parsing it. Additionally, if the network is quite slow then your application will actually spend most of its time idle, waiting for the next data packet. That there is *prime CPU real estate*, which could be put to better use by parsing and handling the data as it arrives. The way to do this is to use NSInputStream to fetch the data, but unless you dive down into the CoreFoundation-based CFNetwork API (which is all pure C), then you're stuck with NSURLConnection's data-received callbacks. Aren't you?

Luckily, it's relatively easy to create a simple stream class, as you saw in the previous chapter, and therefore it's possible to create an NSInputStream subclass that serves up the data provided by an NSURLConnection. In fact, one of your authors has already done so[4]. The basic principle works like this:

- Initialize your input stream subclass using an NSURLRequest. Create an NSURLConnection here, but do not start it until the stream is opened.

[3]There's a video of this: WWDC 2010 in iTunes U, Session 117. Dave's section begins at the 27-minute mark.

[4]See http://github.com/AlanQuatermain/AQURLConnectionInputStream

- Make the request and response objects available via custom properties on the stream (accessible via NSStream's -propertyForKey: API).
- Don't send NSStreamEventOpenCompleted to the stream's delegate until either -connection:didReceiveResponse: or -connection:didReceiveData: has been called for the first time.
- Send any data received in -connection:didReceiveData: straight to the input stream's delegate by sending NSStreamEventHasBytesAvailable. Continue sending that event until the entire data packet has been consumed, or until the delegate chooses not to read any data (i.e., it's waiting for a larger packet).
- When data arrives, place it into an instance variable (or copy it into a special buffer) so it's accessible from -read:maxLength: and -getBuffer:length:.
- Since NSURLConnection has its own methods to be scheduled with an NSRunLoop, your implementation of the schedule/remove methods will just call straight through to your connection's implementation.
- Since the URL connection's delegation methods have already been dispatched to the run-loop and mode selected by the stream's client, you can just call the delegate's -stream:handleEvent: method directly, so no messing about with run-loop sources.

Most of this is fairly straightforward, but for illustration Listing 6-6 shows the implementation of data handling in such a stream.

Listing 6-6. Farming out NSURLConnection Data as an NSInputStream

```
- (void) connection: (NSURLConnection *) connection didReceiveData: (NSData *) data
{
    if ( _streamStatus == NSStreamStatusOpening )
    {
        _streamStatus = NSStreamStatusOpen;
        [_delegate stream: self handleEvent: NSStreamEventOpenCompleted];
    }

    [_buffer appendData: data];
    NSUInteger bufLen = 0;

    // we break when we run out of data, are closed, or when the delegate
    // doesn't read any bytes
    do
    {
        bufLen = [_buffer length];
        [_delegate stream: self handleEvent: NSStreamEventHasBytesAvailable];

    } while ( (bufLen != [_buffer length]) &&
              (_streamStatus == NSStreamStatusOpen) &&
              ([_buffer length] > 0) );
}
```

Network Data

Data sent across the network can be anything, but the most common formats currently used are JSON (JavaScript Object Notation) and XML (eXtensible Markup Language). The `Foundation` framework on OS X and iOS provides facilities for the handling of both types of data.

Reading and Writing JSON

The `Foundation` framework provides the `NSJSONSerialization` class for working with JSON. This class converts between JSON string data and property list types such as dictionaries, arrays, strings, and numbers. Figure 6-2 shows how the data is mapped to different types.

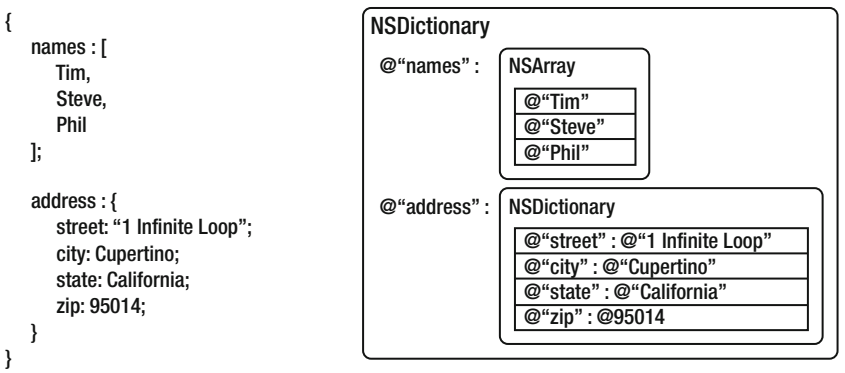

***Figure 6-2.** An example of equivalent JSON and property list types*

When reading JSON data, all keyed lists (elements between { and } characters) are turned into `NSDictionary` instances, and all flat arrays (elements between [and] characters) are turned into `NSArray` instances. Any strings encountered along with the names of named items in keyed lists are turned into `NSString`s, and all purely numeric strings (except for keys) are represented as `NSNumber`s. Lastly, any values of `null` are represented using `NSNull`.

The same thing applies if you're going the other way, of course: any array or dictionary can be transformed into JSON data, provided it only contains the objects specified above. The `jsonize` sample project (available in its entirety online) converts between property list and JSON formats. Its core conversion method is shown in Listing 6-7.

Listing 6-7. Converting Between JSON and Property List Formats

```
NSData * converted = nil;
if ( inputIsJSON )
{
    id obj = [NSJSONSerialization JSONObjectWithData: inputData options: 0
                                               error: &error];
    if ( obj != nil )
    {
        converted = [NSPropertyListSerialization dataWithPropertyList: obj
                         format: NSPropertyListXMLFormat_v1_0 options: 0 error: &error];
    }
}
else
{
    id obj = [NSPropertyListSerialization propertyListWithData: inputData
                                            options: 0 format: NULL error: &error];
    if ( obj != nil )
    {
        if ( [NSJSONSerialization isValidJSONObject: obj] == NO )
        {
            ...
        }

        converted = [NSJSONSerialization dataWithJSONObject: obj
                                      options: NSJSONWritingPrettyPrinted error: &error];
    }
}
```

> **Tip** To quickly find a built application in Xcode, pop open the Products group in the Project Navigator, right-click (or Control-click) the built application in there, and choose "Show in Finder" (the topmost option) in the pop-up menu. Then you can open the Terminal application, type `cd` and a space, and drag the containing folder (in this case `Debug`) from the Finder onto the Terminal window to have its path entered for you. Press Enter and you'll be ready to run the application.

It's also possible to both read and write JSON data (and property list data) to or from a stream. This means that you can create an `NSInputStream` wrapping your connection as I showed earlier and hand that directly to the `NSJSONSerialization` API to have it handled. Of course, the API in question is a synchronous call, which means it's not exactly perfect, but it at least solves the "download all before parsing" issue discussed above.

Working with XML

On OS X there are two APIs available to work with generic XML data. There's a tree-based parser, which creates a hierarchical structure matching that of the XML document; this is rooted in the `NSXMLDocument` class. There's also an event-based parser, implemented by the `NSXMLParser` class. On iOS, you only have the event-based parser, and with a little profiling it's not difficult to

see why that decision was made. For example, using `NSXMLParser`'s stream API to parse a 23MB XML file on OS X, the system allocated an additional 70-80MB of memory while parsing, and working based on a single data object used an extra 130MB of memory. `NSXMLDocument`, on the other hand, allocated a whopping *370MB* of memory!

The result for the streaming parser seems quite high here, but it's ultimately affected by the memory available in the system; on a laptop there's plenty of room to expand and consume the data as fast as possible, while on a mobile device such as an iPhone there's much less room and the data will be handled in smaller chunks as a result. In Jim's parser, the data is always handled in 4KB chunks, with the result that it uses a grand total of 68 *kilobytes* of data while running[5]. It takes a bit longer though, so be aware of the trade-off.

The following examples use the XML document from Listing 6-8 to illustrate the various concepts.

Listing 6-8. A Sample XML Document

```
<?xml version="1.0"?>
<CATALOG>
        <PLANT>
                <COMMON LANG="EN">Bloodroot</COMMON>
                <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
                <ZONE>4</ZONE>
                <LIGHT>Mostly Shady</LIGHT>
                <PRICE>2.44</PRICE>
                <AVAILABILITY>031599</AVAILABILITY>
        </PLANT>
        <PLANT>
                <COMMON LANG="EN">Columbine</COMMON>
                <BOTANICAL>Aquilegia canadensis</BOTANICAL>
                <ZONE>3</ZONE>
                <LIGHT>Mostly Shady</LIGHT>
                <PRICE>9.37</PRICE>
                <AVAILABILITY>030699</AVAILABILITY>
        </PLANT>
        <PLANT>
                <COMMON LANG="EN">Marsh Marigold</COMMON>
                <BOTANICAL>Caltha palustris</BOTANICAL>
                <ZONE>4</ZONE>
                <LIGHT>Mostly Sunny</LIGHT>
                <PRICE>6.81</PRICE>
                <AVAILABILITY>051799</AVAILABILITY>
        </PLANT>
</CATALOG>
```

[5]More information can be found online here:
`http://alanquatermain.me/post/93664991/aqxmlparser-test-redux`

XML Trees

An XML tree is composed of *nodes*, represented on OS X by the `NSXMLNode` class. Any piece of data within an XML document is a node, including the tags, any text, processing instructions, and the document itself. A tag pair and its contents is an *element*, represented by `NSXMLElement`, and only elements can contain other nodes. In Listing 6-8, for example, the root node is a `CATALOG` element, and this contains three `PLANT` elements and some text nodes. Don't see the text nodes? Oh, they're there all right—all that whitespace used to make the source look pleasant is still included in the document. For example, the first text node within the `CATALOG` element consists of two characters: a newline and a tab character. Following that is the first `PLANT` element followed by another newline/tab pair, and so on. As a result, it is often much better to serve up XML data in as compressed a format as possible, to avoid having your tree (and thus your memory!) cluttered up with text node objects for data in which you're not at all interested.

Another approach is to take advantage of the fact that whitespace within an XML tag itself is ignored by placing the newlines and tabs within the tags themselves. This produces XML similar to that seen in Listing 6-9. Since there is no space between the closing and opening braces used to delimit the tags, there are no extra text nodes in the XML tree. It looks a little strange, perhaps , but it's both structured and clean.

Listing 6-9. A Nicely Formatted XML Document with a Compact Tree Structure

```
<CATALOG
        ><PLANT
                ><COMMON LANG="EN">Bloodroot</COMMON
                ><BOTANICAL>Sanguinaria canadensis</BOTANICAL
                ><ZONE>4</ZONE
                ><LIGHT>Mostly Shady</LIGHT
                ><PRICE>2.44</PRICE
                ><AVAILABILITY>031599</AVAILABILITY
        </PLANT
></CATALOG>
```

XPath and XQuery

The primary advantage of tree-based parsers is that the generated structure is easy to work with programmatically. The most common way to work with structured XML documents is to use XPath or XQuery expressions. These provide a way to simply access groups of data within a larger XML document and are used by calling the `NSXMLNode` instance methods `-nodesForXPath:error:`, `-objectsForXQuery:error:` or `-objectsForXQuery:constants:error:`.

XPaths, as their name suggests, are based around the same concept as filesystem paths. The same as filesystem paths, *absolute* XPaths begin with a forward-slash character — `/`. Relative XPaths omit this leading character.

An XPath expression is applied to a particular node in the XML tree. When searching a document you will often supply an absolute XPath or perform the search based on the root node or the `NSXMLDocument` itself. The XPath specification does also supply a number of

functions and special selectors to assist in tree traversal. You can see a few examples of XPaths in Listing 6-10 designed to work on the XML from Listing 6-8. Can you determine what they're selecting just by reading them?

Listing 6-10. Some Example XPaths

```
/CATALOG/PLANT/COMMON

/CATALOG/PLANT/BOTANICAL/text()

/CATALOG/PLANT[ZONE>3]/COMMON

//COMMON[@LANG='EN']
```

The first example selects every `COMMON` element in the document that is a child of a `PLANT` element, which is a child of a `CATALOG` element, which is the root of the document. The second locates every `BOTANICAL` element in the same position, but returns the text content of that element (not the text *node* within the element, but the actual raw, unprocessed characters that fall between the element's opening and closing tags).

The third example specifies that it should only match `PLANT` elements that have a `ZONE` sub-element, and further that only those whose `ZONE` element contains a numeric value greater than 3. It will then select and return the `COMMON` elements within those matched `PLANT` elements.

The last example has two differences: first of all, it begins with two path delimiters, which means that the parser can match the first element specified anywhere in the element tree, regardless of its parents. The statement in the square brackets is similar to that in the third example, but by placing an ampersat (@) at the start of the name you state that you're comparing not a sub-element's value but the value of one of this element's attributes. Since you're comparing the value against a string, the value is enclosed in single quotes to indicate that.

XQuery builds a scripting language on top of the XPath syntax; it essentially enables you to query the contents of an XML document in the way you would query an SQL database. It's quite powerful and wider in scope than I can devote time to here, but Listing 6-11 contains a short example query to whet your appetite. The query itself finds all `PLANT` elements whose `PRICE` sub-element's value is greater than five and returns the `COMMON` sub-elements, sorted into ascending order based on the `COMMON` sub-element's value. If you're familiar with the capabilities of SQL queries, then you have an idea as to how this could be used to interact with quite complex XML documents.

Listing 6-11. A Sample XQuery Expression

```
For $x in /CATALOG/PLANT
Where $x/PRICE > 5
Order by $x/COMMON
Return $x/COMMON
```

Event-Based XML Handling

Handling XML with the `NSXMLParser` class is based entirely on event messages sent to the parser's delegate. The parser reads and interprets the XML document in a linear fashion and notifies its delegate of each type of content it encounters, whether opening tags, closing tags, characters, CDATA (raw Character Data, not parsed or interpreted), entities, processing

instructions, or others. There are quite a few. Most of the time, however, you'll be interested in only a few of them:

- Opening tags and their attributes, provided through the `-parser:didStartElement:namespaceURI:qualifiedName:attributes:` method.
- Closing tags, provided through `-parser:didEndElement:namespaceURI:qualifiedName:`.
- Text, provided through `-parser:foundCharacters:`; and raw CDATA blocks, provided through `-parser:foundCDATA:`.

The delegate can also be given details of any fatal parsing or validation errors encountered, but since these errors are returned through `NSXMLParser`'s `-parserError` method, it's not common to implement the delegation methods for them.

Now, these are long method names, and if you're inspecting a large document with many different tags, each of which has its own handling, you're looking at a lot of `if` statements inside your element handlers, which isn't ideal. Rather than do that, I suggest an alternative approach that makes use of Objective-C's dynamic type system and message dispatch.

The aim is to create delegate classes that implement some simple methods to handle start and/or end tags, named according to some predetermined rules. Since opening tags contain an element's attributes, you'll define your handler for those as '`-start<Element>WithAttributes:(NSDictionary*)attributes`', where `<Element>` will be replaced with the real element's name, with its first character uppercased. Thus you will have methods like `-startTitleWithAttributes:` for `<title>` tags, and `-startPLANTWithAttributes:` for `<PLANT>` tags (remember that XML is case-sensitive). For ending tags you'll simply use `-end<Element>`, thus `-endTitle` or `-endPLANT`.

The magic behind this will be a shared superclass, one that will conform to the `NSXMLParserDelegate` protocol. In its start and end element handlers it will build a selector name as a string and check with the runtime to see if it implements that method. If so, it calls the method (passing attributes to start tag handlers). This superclass will also handle text and CDATA by accumulating it within an `NSString` instance variable, which will be cleared each time a new start or end tag for an element is encountered. This will be made available to subclasses via a property. The handlers for text and CDATA blocks can be seen in Listing 6-12.

Listing 6-12. Handling Text and CDATA

```
- (void)parser:(NSXMLParser*)parser foundCharacters:(NSString*)string
{
    if ( [string length] == 0 )
        return;

    @autoreleasepool {
        if ( self.characters != nil )
            self.characters = [self.characters stringByAppendingString:string];
        else
            self.characters = string;
    }
}
```

```
- (void)parser:(NSXMLParser*)parser foundCDATA:(NSData*)CDATABlock
{
    if ( [CDATABlock length] == 0 )
        return;

    NSString *str = [[NSString alloc] initWithData:CDATABlock
                                          encoding:NSUTF8StringEncoding];
    [self parser:parser foundCharacters:str];
}
```

The handling of elements' start and end tags is a little more involved, but ultimately still fairly simple to understand. An important consideration is that any element names that contain hyphens will not be usable as-is; you could replace hyphens with underscores, but I have chosen to convert them to CamelCase to match the rest of the method definition. You can see an implementation of this in Listing 6-13.

Listing 6-13. Handling Element Start and End Tags

```
- (NSString*)selectorCleanNameForElementName:(NSString*)elementName
{
    NSMutableString *selStr = [NSMutableString new];

    @autoreleasepool {
        // ensure the first character of the element name is uppercased
        [selStr appendString: [[elementName substringToIndex:1] uppercaseString]];

        // append the rest of the string (provided it's not 1-character long)
        if ( [elementName length] > 1 )
            [selStr appendString: [elementName substringFromIndex:1]];

        // handle hyphens properly
        NSRange range = [selStr rangeOfString:@"-"];
        for ( ; range.location != NSNotFound; range = [selStr rangeOfString:@"-"] )
        {
            NSString *cap = [[selStr substringWithRange:NSMakeRange(range.location+1,1)]
                                          uppercaseString];
            range.length += 1;
            [selStr replaceCharactersInRange:range withString:cap];
        }
    }

    return ( selStr );
}

- (void)parser:(NSXMLParser*)parser didStartElement:(NSString*)elementName
      namespaceURI:(NSString*)namespaceURI qualifiedName:(NSString*)qName
        attributes:(NSDictionary*)attributes
{
    @autoreleasepool {
        NSString *cleanName = [self selectorCleanNameForElementName:elementName];
        NSString *selStr = [NSString stringWithFormat:@"start%@WithAttributes:",
                                                                    cleanName];
```

```
        SEL selector = NSSelectorFromString(selStr);
        if ( [self respondsToSelector:selector] )
            [self performSelector:selector withObject:attributes];

        self.characters = nil;
    }
}

- (void)parser:(NSXMLParser*)parser didEndElement:(NSString*)elementName
      namespaceURI:(NSString*)namespaceURI qualifiedName:(NSString*)qName
{
    @autoreleasepool {
        NSString *cleanName = [self selectorCleanNameForElementName:elementName];
        NSString *selStr = [NSString stringWithFormat:@"end%@", cleanName];

        SEL selector = NSSelectorFromString(selStr);
        if ( [self respondsToSelector:selector] )
            [self performSelector:selector];

        self.characters = nil;
    }
}
```

Network Service Location

Back in the days of yore (a.k.a. the 1990s) Apple computers used a suite of protocols called AppleTalk to communicate on a local network (or even further!). While the main networking protocol hasn't aged well and is very rarely used today, the suite had some very useful features, not the least of which was the Name Binding Protocol (NBP), which had the capability to advertise the availability of services across the network. In the world of the Internet Protocol and Ethernet, however, nothing similar existed. To connect to a web server, you needed to know its address and you would connect to port 80. If there was another web server running on a different port, someone would have to tell you about it explicitly so that you might connect. The Domain Name System (DNS) offers a means of obtaining an IP address based on a name such as `www.apple.com`. This normally requires a dedicated server and manual configuration, however, which makes it a poor solution for non-technical users.

In 2002, Apple engineer Stuart Cheshire designed and published a new standard through the Internet Engineering Task Force entitled *DNS Service Discovery*. This standard described various updates to the DNS system and protocol, adding support for the resolution of information on individual services on a system to their individual port numbers. It also added a protocol through which a query to a DNS service could remain active, receiving updates whenever changes occurred without needing to reconnect and poll the server continually. This was then married to the *multicast DNS* (mDNS) system built into OS X 10.2 to allow all hosts on a local area network to provide information on any services available on that host. The result was named Rendezvous, although its name was sadly changed to the less appropriate Bonjour[6] two years and one trademark lawsuit later.

The Bonjour service is available in both OS X and iOS, implemented at a very low level. Luckily for you, it has a series of higher-level APIs available right up to the Cocoa layer, allowing you to

[6]Oh yes, I went there.

not only find services on the local network but also advertise your own in only a few easy steps.

Service Resolution

You can search the network for services of a given type using an instance of the `NSNetServiceBrowser` class. The service browser object operates asynchronously and sends messages to a delegate object on a given run loop as it encounters various events. A browser can be used to search for available domains either for browsing or registering new services (a domain may be browsable but not support dynamic addition of new items), or it can search a domain for services. The browser's delegate can implement a number of different methods to receive updates during the search process, which are outlined in Table 6-2.

Table 6-2. NSNetServiceBrowser Delegate Methods

Message	Description
`- (void) netServiceBrowserWillSearch: (NSNetServiceBrowser*)browser`	The browser sends this message to its delegate before it begins any form of search operation. If the search cannot start, the delegate will instead receive a `-netServiceBrowser:didNotSearch:` message.
`- (void) netServiceBrowserDidStopSearch: (NSNetServiceBrowser*)browser`	Sent to the browser's delegate whenever a search stops running, whether because all results have been received or because the browser was explicitly stopped.
`- (void) netServiceBrowser:(NSNetServiceBrowser*)browser didNotSearch:(NSDictionary*)errorDict`	When the browser is unable to begin a search for some reason, it will send this message to its delegate. The supplied dictionary contains two key/value pairs describing the error domain and code.
`- (void) netServiceBrowser:(NSNetServiceBrowser*)browser didFindDomain:(NSString*)domain moreComing:(BOOL)moreComing`	Each domain discovered during a search for browse or register domains will be reported to this method. If the passed domain is the last one, `moreComing` will be `NO`.
`- (void) netServiceBrowser:(NSNetServiceBrowser*)browser didFindService:(NSNetService*)service moreComing:(BOOL)moreComing`	Each service of the requested type found by the browser is reported to this method. When `moreComing` is `NO`, all results have been processed.
`- (void) netServiceBrowser:(NSNetServiceBrowser*)browser didRemoveDomain:(NSString*)domain moreComing:(BOOL)moreComing`	When a domain becomes unavailable, your delegate object will be notified through this method.
`- (void) netServiceBrowser:(NSNetServiceBrowser*)browser didRemoveService:(NSNetService*)service moreComing:(BOOL)moreComing`	When a service becomes unavailable, this method will be called to notify your delegate object of the change.

The NSNetServiceBrowser class needs to have a delegate set and be scheduled in a run-loop before you can receive these methods. Use -setDelegate: and -scheduleInRunLoop:forMode: to make this happen.

Once you've attached your browser to a run-loop, you can search for domains using either -searchForBrowsableDomains or -searchForRegistrationDomains. You can search for services by specifying a type and domain to -searchForServicesOfType:inDomain:. For most purposes, however, you won't need to search for domains yourself; when you search for a service, simply pass an empty string as the domain parameter and the browser will automatically search all appropriate domains. This typically includes the local domain, which refers to the link-local network (everything on the same side of a network router), but may also include an iCloud domain (for example, nnnnnn.members.btmm.icloud.com) if the user has an iCloud account. This is how features such as Back To My Mac are able to display your home computer in the Finder while at work—because it's registered with your account's corresponding iCloud domain, which is searched by default by NSNetServiceBrowser.

Service types are designed to work along with the host and domain components of the DNS standard and are thus nested with their transport types. The idea is that a host is given the address host.domain.com and services on that host are given the address service.transport.host.domain.com. By convention all address components that refer to items within a single host use names that begin with underscore characters. The services themselves are named based on their corresponding registered names as recorded in your computers services file (located at /etc/services), and they ordinarily use the transports mentioned in that file. Transports are again prefixed with an underscore character. Some examples of common service types can be seen in Table 6-3.

***Table 6-3.** Common DNS Service Types*

Service Type	Description
_http._tcp	Web servers, which operate using the HyperText Transfer Protocol over TCP.
_ftp._tcp	File Transfer Protocol servers.
_afpovertcp._tcp	The AppleShare Filing Protocol (the standard means of accessing another computer's files on the Mac), in its TCP/IP implementation.
_daap._tcp	iTunes libraries, shared on the network using the Digital Audio Access Protocol.

So, to search for available web servers, you would specify a service type of _http._tcp and an empty domain. Listing 6-14 shows how you might go about doing this.

***Listing 6-14.** Searching for Web Servers*

```
NSNetServiceBrowser *browser = [NSNetServiceBrowser new];
[browser setDelegate:self];
[browser scheduleInRunLoop:[NSRunLoop mainRunLoop] forMode:NSRunLoopCommonModes];
[browser searchForServicesOfType:@"_http._tcp" inDomain:@""];
```

When the browser returns services to you, those services have a name and type but any other details have not yet been resolved. To do so, you need to provide a delegate to the given

NSNetService instance and implement the -netServiceDidResolveAddress: method in that delegate to find out when the information becomes available. You then ask the service to resolve by sending it the -resolveWithTimeout: method, as shown in Listing 6-15.

Listing 6-15. Resolving an NSNetService

```
- (void)netServiceBrowser:(NSNetServiceBrowser*)browser
           didFindService:(NSNetService*)service
               moreComing:(BOOL)moreComing
{
    [service setDelegate:self];
    [service scheduleInRunLoop:[NSRunLoop mainRunLoop] forMode:NSRunLoopCommonModes];
    [service resolveWithTimeout:5.0];
}

- (void)netServiceDidResolveAddress:(NSNetService*)sender
{
    NSLog(@"Service %@ resolved", [sender name]);
    NSLog(@"  - host name: %@", [sender hostName]);
    NSLog(@"  - port: %ld", [sender port]);
    NSLog(@"  - IP v4 Address: %@", [self IP4Address:[sender addresses]]);
    NSLog(@"  - IP v6 Address: %@", [self IP6Address:[sender addresses]]);
}
```

The -addresses method of NSNetService will return an array of data objects encapsulating low-level socket addresses, which you can use to create connected sockets using C-based methods from either the POSIX or CoreFoundation layers. If you'd like to stay at the Foundation layer, you can call -getInputStream:outputStream: to receive by reference a pair of streams ready to open and communicate with the service.

Publishing a Service

Service publishing is done directly through the NSNetService class itself. Having created a service and made it available for incoming connections, the details you want to provide are you service's type, name, and the port on which you are listening for new connections. You provide this to a new instance of NSNetService in its registration initializer, -initWithDomain:type:name:port:. As when browsing services, passing an empty string in the domain parameter will register your service in all available domains. You then publish service's information for others to find by sending the -publish or -publishWithOptions: message to your NSNetService instance. When you're done, you notify the rest of the network that your service is no longer available by sending -stop.

In order to receive information about the status of your service's DNS record publication, you can provide a delegate object. When publishing a service record, the messages from Table 6-4 are sent to the delegate.

Table 6-4. NSNetService Publication Delegate Messages

Message	Description
`- (void)netServiceWillPublish:(NSNetService*)service`	Sent to the delegate before the publish operation is due to start.
`- (void)netServiceDidPublish:(NSNetService*)service`	Notifies the delegate that the service was successfully published.
`- (void) netService:(NSNetService*)service didNotPublish:(NSDictionary*)errorInfo`	Provides error information if a publish operation failed for any reason.

The provided sample projects `NSServiceBrowser` and `NSServiceVendor` show some more in-depth examples of the use of services. As an example, you can use the `VendService` command-line application to vend a service with a given name and type, and then use the `NSServiceBrowser` graphical application to look for it on the network. You can then shut down and start up more services of the same type and see them appear and disappear in the browser.

Summary

I hope this chapter was a lot simpler to digest than earlier chapters; part of this is because of the groundwork you've done leading up to this point—on the shoulders of giants you stand. The main lesson from this chapter is the correct use of the available asynchronous networking APIs in the `Foundation` framework. Here's a summary of the key points:

- Always use asynchronous methods. I am *so* not kidding on this one.
- Handle network URLs properly through the judicious use of `NSURLRequest` and `NSURLConnection`.
- Authenticate access to network resources in a protocol-agnostic manner through `NSURLConnection`'s authentication delegation.
- Process network data in JSON and XML formats. Now you know why iOS doesn't include any tree-based XML parsers by default.
- Parse network data as it arrives.
- Use XPath and XQuery on those tree-based XML parsers.
- Discover network services using `NSNetServiceBrowser` and vending your own with `NSNetService`.

It is my hope that the core lessons here will remain firmly entrenched in your mind for the remainder of your career.