Networking Overview

Developer

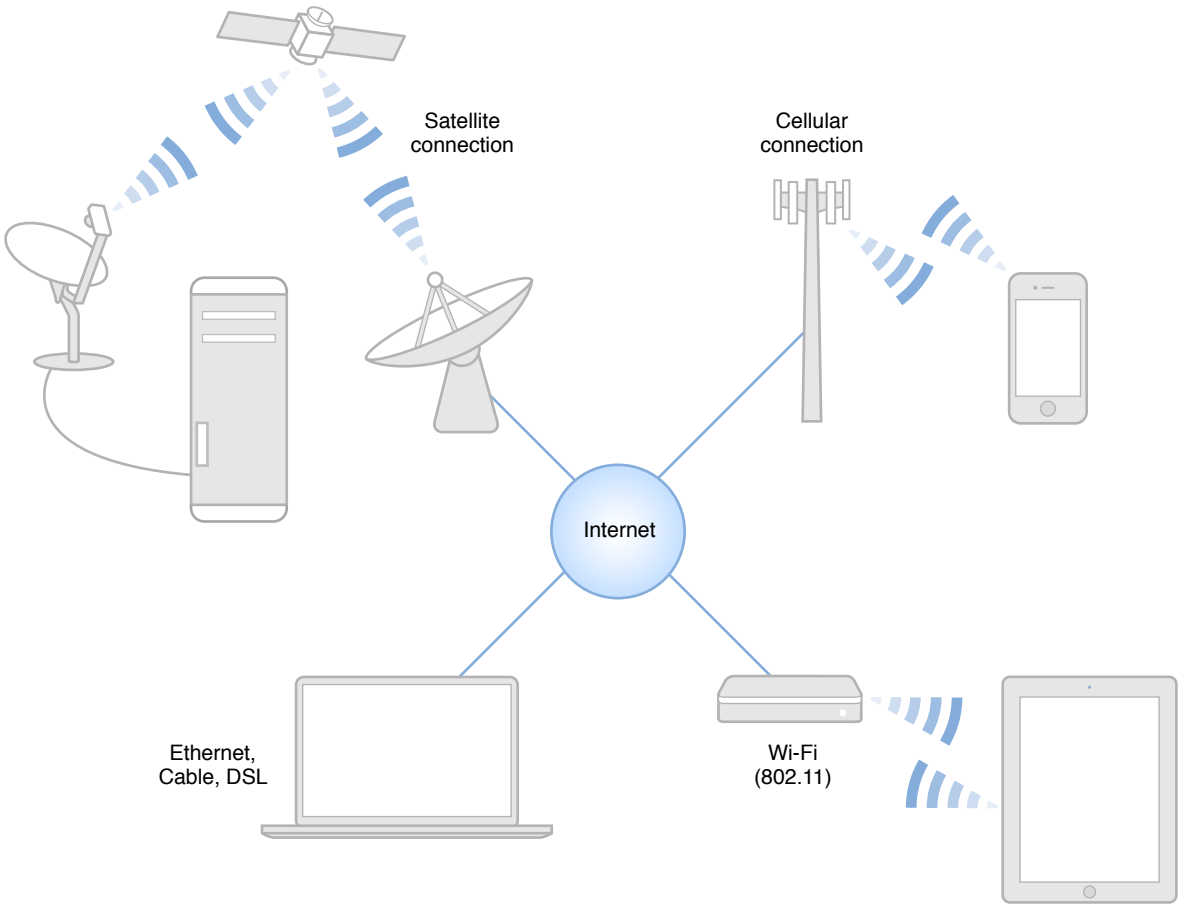
Contents

Figures, Tables, and Listings

About Networking

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

The world of networking is complex. Users can connect to the Internet using a wide range of technologies—cable modems, DSL, Wi-Fi, cellular connections, satellite uplinks, Ethernet, and even traditional acoustic modems. Each of these connections has distinct characteristics, including differences in bandwidth, latency, packet loss, and reliability.



To add further complexity, the user's connection to the Internet does not tell the whole story. On its way from the user to an Internet server, the user's network data passes through anywhere from one to dozens of physical interconnects, any one of which could be a high-speed OC-768 line (at almost 40 billion bits per second), a meager 300 baud modem (at 300 bits per second), or anything in-between. Worse, at any moment, the speed of the user's connection to a server could change drastically—someone could turn on a microwave oven that interferes with the user's Wi-Fi communications, the user could walk or drive out of cellular range, someone on the other side of the world could start downloading a large movie from the server that the user is trying to access, and so on.

As a developer of network-based software, your code must be able to adapt to changing network conditions, including performance, availability, and reliability. This document tells you how.

At a Glance

Networks are inherently unreliable—cellular networks doubly so. As a result, good networking code tends to be somewhat complex. Among other things, your software should:

- **Transfer only as much data as required to accomplish a task.** Minimizing the amount of data sent and received prolongs battery life, and may reduce the cost for users on metered Internet connections that bill by the megabyte.
- **Avoid timeouts whenever possible.** You probably don't want a webpage to stop loading just because the loading process took too long. Instead, provide a way for the user to cancel the operation.

  In certain rare situations, data becomes irrelevant if delayed substantially. In these situations, it may make sense to use a protocol that does not retransmit packets. For example, if you are writing a real-time multiplayer game that sends tiny state messages to another device over a local area network (LAN) or Bluetooth, it is often better to miss a message and make assumptions about what is happening on the other device than to allow the operating system to queue those packets and deliver them all at once. For most purposes, however, unless you have to maintain compatibility with existing protocols, you should generally use TCP.
- **Design user interfaces that allow the user to easily cancel transactions that are taking too long to complete.** If your app performs downloads of potentially large files, you should also provide a way to pause those downloads and resume them later.
- **Handle failures gracefully.** A connection might fail for any number of reasons—the network might be unavailable, a hostname might not resolve successfully, and so on. When failures occur, your program should continue to function to the maximum degree possible in an offline state.

To add further complexity, sometimes a user may have access to resources only while on certain networks. For example, AirPlay can connect to an Apple TV only while on the same network. Corporate network resources can be accessed only while at work or over a virtual private network (VPN). Visual Voicemail may be accessible only over the cellular carrier's network (depending on the carrier). And so on.

In particular, you should avoid interfaces that require the user to babysit your program when the network is malfunctioning. Don't display modal dialogs to tell the user that the network is down. Do retry automatically when the network is working again. Don't alert the user to connection failures that the user did not initiate.

- **Degrade gracefully when network performance is slow.** Because the bandwidth between the user's device and his or her ISP is limited, your app can reach other devices on the user's home network much more quickly than servers on the other side of the world. This difference becomes even greater when someone else on the local network starts using that limited bandwidth for other purposes.
- **Choose APIs that are appropriate for the task.** If there is a high-level API that can meet your needs, use it instead of rolling your own implementation using low-level APIs. If there is an API specific to what you are doing (such as a game-centric API), use it.

  By using the highest-level API, you are providing the operating system with more information about what you are actually trying to accomplish so that it can more optimally handle your request. These higher-level APIs also solve many of the most complex and difficult networking problems for you—caching, proxies, choosing from among multiple IP addresses for a host, and so on. If you write your own low-level code to perform the same tasks, you have to handle that complexity yourself (and debug and maintain the code in question).
- **Design your software carefully to minimize security risks.** Take advantage of security technologies such as Secure Sockets Layer (SSL) and Transport Layer Security (TLS) to prevent spoofing and hide sensitive data from prying eyes, and scrutinize untrusted content to prevent buffer and integer overflows.

This document will help you learn these concepts and more.

Learn Why Networking Is Hard

Although writing networking code can be easy, for all but the most trivial networking needs, writing *good* networking code is not. Depending on your software's needs, it may need to adapt to changing network performance, dropped network connections, connection failures, and other problems caused by the inherent unreliability of the Internet itself.

Relevant Chapter: Designing for Real-World Networks (page 13)

OS X and iOS Provide APIs at Many Levels

You can accomplish the following networking tasks in both OS X and iOS with identical or nearly identical code:

- Performing HTTP/HTTPS requests, such as GET and POST requests
- Establishing a connection to a remote host, with or without encryption or authentication
- Listening for incoming connections
- Sending and receiving data with connectionless protocols
- Publishing, browsing, and resolving network services with Bonjour

Relevant Chapters: Assessing Your Networking Needs (page 21)

Discovering and Advertising Network Services (page 24)

Making HTTP and HTTPS Requests (page 31)

Using Sockets and Socket Streams (page 39)

Secure Communication Is Your Responsibility

Proper networking security is a necessity. You should treat all data sent by your user as confidential and protect it accordingly. In particular, you should encrypt it during transit and protect against sending it to the wrong person or server.

Most OS X and iOS networking APIs provide easy integration with TLS for this purpose. TLS is the successor to the SSL protocol. In addition to encrypting data over the wire, TLS authenticates a server with a certificate to prevent spoofing.

Your server should also take steps to authenticate the client. This authentication could be as simple as a password or as complex as a hardware authentication token, depending on your needs.

Be wary of all incoming data. Any data received from an untrusted source may be a malicious attack. Your app should carefully inspect incoming data and immediately discard anything that looks suspicious.

Relevant Chapter: Using Networking Securely (page 41)

iOS and OS X Offer Platform-Specific Features

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.

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.

Relevant Sections: iOS Requires You to Handle Backgrounding and Specify Cellular Usage Policies (page 46)

OS X Lets You Make Systemwide Changes (page 48)

Networking Must Be Dynamic and Asynchronous

A device's network environment can change at a moment's notice. There are a number of simple (yet devastating) networking mistakes that can adversely affect your app's performance and usability, such as executing synchronous networking code on your program's main thread, failing to handle network changes gracefully, and so on. You can save a lot of time and effort by designing your program to avoid these issues to begin with instead of debugging it later.

Relevant Chapter: Avoiding Common Networking Mistakes (page 49)

How to Use This Document

This document is intended to be read sequentially.

The first chapter, Designing for Real-World Networks (page 13), explains the challenges you will face when writing software that uses networking, why latency matters, and other concepts that you should know before you write the first line of networking code.

The next chapter, Assessing Your Networking Needs (page 21), provides more details about choosing an API family and determining what types of networking tasks your program will perform. This chapter then points you to other chapters (Discovering and Advertising Network Services (page 24), Making HTTP and HTTPS Requests (page 31), and Using Sockets and Socket Streams (page 39)) that describe some common networking tasks that your program might need to perform.

Using Networking Securely (page 41) and Avoiding Common Networking Mistakes (page 49) provide guidance that can help you avoid common networking mistakes.

Finally, Supporting IPv6 DNS64/NAT64 Networks (page 57) explains how to make sure your app is compatible with IPv6-only networks.

See Also

This document is intended as a high-level overview of networking concerns in OS X and iOS. The documents below provide additional depth and breadth.

Learn What's Happening Under the Hood

A basic understanding of the way networks work can help you understand why they behave (or misbehave) as they do. Thus, you should learn at least the basic underlying concepts before you write the first line of code. At minimum, you should be familiar with packets and encapsulation, connection-based versus connectionless protocols, subnets and routing, domain name lookup, bandwidth, and latency. To learn about this subject, read the following document:

- *Networking Concepts*

Learn About Specific Technologies

For more in-depth information, consult one of the following guides for the primary documentation on a particular subject:

- *URL Session Programming Guide*
- *Stream Programming Guide*
- *CFNetwork Programming Guide*
- *NSNetServices and CFNetServices Programming Guide*

Learn How to Share Documents Between OS X and iOS

The following documents describe techniques you can use to share documents between OS X and iOS:

- *iCloud Design Guide*
- *Document Transfer Strategies*

Designing for Real-World Networks

In an ideal world, networking "just works". Your network connection is reliable, fast, and low latency. In the real world, networking works most of the time, but when it breaks, it often breaks in strange and fascinating ways. For example:

- An overloaded or broken network link can exhibit packet loss. If a link loses enough packets, it may be difficult to establish connections across that link, and performance may fall to a tiny fraction of what you would expect.
- When a network link becomes saturated, routers on either side of that link buffer the traffic to avoid losing data. This adds additional latency. It is not uncommon to see latency measured in whole seconds over heavily loaded DSL connections.
- Captive networks (often used in hotels, coffee shops, and other public places) may intercept your software's HTTP requests and provide a login page instead of the expected data.
- Firewalls between the user and the destination may block connections on all but a handful of ports.
- Firewalls that perform network address translation (NAT) may not allow remote servers to connect back to ports on the user's computer or other device.
- Third-party firewall software may block your software's outgoing connection requests for minutes at a time while waiting for the user to grant permission to open the connection.

Although your software cannot magically fix a truly broken network, poorly written networking code can easily make things much, much worse. For example, suppose a server is heavily overloaded and is taking 45 seconds to respond to each request. If your software connects to that server with a 30-second timeout, it contributes to the server's workload, but never successfully receives any data.

And even when the network is working perfectly, poorly written networking code can cause problems for the user—poor battery life, poor performance, and so on. The sections in this chapter describe things that your software should do to minimize users' pain, both when conditions are ideal and when things go wrong.

Using Power And Bandwidth Efficiently

The most important thing to consider when writing networking code is that every time your software uploads or downloads data, it costs the user both time and money.

A network operation costs the user time because:

- The user must wait for the operation to complete before performing some task.
- Data transfers often require wireless radios to remain active. For battery-powered devices, this reduces the amount of time the user can use the device before its battery runs down.

A network operation also costs the user money because bandwidth isn't free. Some costs include:

- Electrical power. Wireless hardware (Wi-Fi, cellular, and so on) consumes a fair amount of power. The longer that wireless hardware is active, the more power it consumes.
- Actual data transferred. Many users (particularly cellular network users) pay for their data based on actual use. The more bytes your software transfers, the more they pay. And even if the user has flat-rate service, the ISP sets that rate based in part on how much bandwidth the average user consumes.
- Bandwidth. Whether the user's network connection is metered by the byte or is a flat-rate service, the user typically pays higher rates for faster connection speeds.

As a developer of networking software, it is your responsibility to minimize the power and bandwidth that your software consumes.

Batch Your Transfers, and Idle Whenever Possible

When writing code in general, to the maximum extent possible, you should perform as much work as you can and then return to an idle state. This applies doubly for network activity. For example:

- If your app streams video clips from an HTTP server, download the entire file at once (or at least a large portion of that file) instead of requesting it a small piece at a time.
- If your app serves advertisements, download several ads at once and show them over a period of time, rather than downloading them as they are needed.
- If your app downloads email messages from a server, download the first few messages at once under the assumption that the user will probably read most of them, rather than downloading each one individually as the user selects it.

Downloading content a bit at a time causes two problems. First, it makes the app more sensitive to minor network delays, causing stalls, video stuttering, and so on. Second, it keeps the cellular or Wi-Fi radio powered up almost continuously. This wastes power, particularly when your app is communicating over a cellular connection. If your app instead downloads lots of data for a brief period of time and then allows the wireless connection to go fully idle, you can significantly improve your users' battery life.

This applies particularly to socket programming. With few exceptions (such as remote terminal programs), you should never send only a few bytes out at a time. Doing so is extremely inefficient in terms of the CPU load, and can cause the operating system to send more packets than necessary.

Download the Smallest Resource Possible, and Cache Resources Locally

Downloading data has many costs associated with it—battery life, performance, and in many cases, actual data transfer costs. For this reason, you should always download the smallest version of an asset that can serve your needs.

For example, if you have an image catalog app that downloads a series of large images and renders them as small thumbnails, you should render those thumbnails on the server. Your app should download only the thumbnail initially, waiting to download the full-size version of an image until the user selects its thumbnail. There are two reasons to do this:

- Transferring data consumes power by keeping the networking hardware and the CPU powered up for longer periods of time. By decreasing the size of the assets your program transfers can improve your users' battery life (assuming that this results in a net decrease in total data transferred, on average).
- If your users are on a metered Internet connection (such as a cellular phone), transferring smaller assets can also reduce your users' data bills.

For the same reason, keeping a local cache of download resources can save time, bandwidth, and battery life. To do this, instead of asking the server for a resource, ask whether that resource has changed since you downloaded it; if it has not, use the local copy.

A number of higher-level APIs in OS X and iOS (`NSURL`, for example) provide support for caching (`NSURLCache`, for example). However, you must choose appropriate sizes for the caches. Whether you are using a built-in caching API or are creating your own, you should experiment with cache sizes and replacement policies to determine what makes the most sense for your app.

Note: There is often a conflict between this goal and the previous goal—downloading lots of data at once so that the network hardware can become idle.

For example, consider an application that loads a gallery of image thumbnails. If the average user scrolls through several screens filled with thumbnails, the application should download enough images to fill the first few screens in a single batch so that the network hardware can become idle between downloads. On the other hand, if the average user never scrolls to the second screen, all of those additional images are wasted bandwidth.

Each networking application must strike a balance between these conflicting goals, and it is up to you, the developer, to decide how best to do so.

Handling Network Problems Gracefully

In today's highly mobile world, you can no longer assume that Internet connectivity, once established, will remain established, or that bandwidth will never increase or decrease—as it is said, change is the only constant. As a developer, you must plan for these common failures and design your code to handle them appropriately.

Design for Variable Network Interface Availability

Network interface availability can change regularly for countless reasons, particularly in iOS. For example, the user could:

- Be traveling on a subway, acquiring a wireless signal at every stop and losing the signal with every departure.
- Move outside the range of the current Wi-Fi network.
- Activate Airplane Mode or turn off Wi-Fi.
- Unplug a network cable.

Because of this, when writing software that uses the network, you must be prepared for network failures. When a network error occurs, your program should decide what to do based on a number of considerations—most importantly, whether the request was made explicitly by the user or not.

For requests made at the user's behest:

- Always attempt to make a connection. Do not attempt to guess whether network service is available, and do not cache that determination.
- If a connection fails, use the `SCNetworkReachability` API to help diagnose the cause of the failure. Then:
  - If the connection failed because of a transient error, try making the connection again.

- If the connection failed because the host is unreachable, wait for the `SCNetworkReachability` API to call your registered callback. When the host becomes reachable again, your app should retry the connection attempt automatically without user intervention (unless the user has taken some action to cancel the request, such as closing the browser window or clicking a cancel button).

- Try to display connection status information in a non-modal way. However, if you must display an error dialog, be sure that it does not interfere with your app's ability to retry automatically when the remote host becomes reachable again. Dismiss the dialog automatically when the host becomes reachable again.

For requests made in the background:

- Attempt to make a connection.

  If desired, use `SCNetworkReachability` to avoid making the connection at inconvenient times—for example, avoiding unnecessary traffic over a cellular connection by checking for the `kSCNetworkReachabilityFlagsIsWWAN` flag.

  > **Important:** Checking the reachability flag does not guarantee that your traffic will never be sent over a cellular connection. See Restrict Cellular Networking Correctly (page 46) for details.

- If the connection fails, use the `SCNetworkReachability` API to wait for the host to become reachable again, then retry your request if it is still useful to do so.

- Do not display any dialog; users generally do not care about failures in background downloads that they did not initiate.

- Avoid retrying too quickly even when the network reachability APIs tell your application that the network has changed. When connections fail repeatedly, you should gradually increase the amount of time you wait between attempts until you reach a reasonably long retry interval (15 minutes, for example).

Your program should be able to respond gracefully to changes in the current network interface. To support this, use the `SCNetworkReachability` API. By registering for network change notifications, your program is alerted when the available network interfaces change.

The *Reachability* sample code demonstrates registering a callback for notification when the current network interface changes. Read *SCNetworkReachability Reference* for a complete discussion of the `SCNetworkReachability` API.

> **Important:** The `SCNetworkReachability` API is not intended for use as a preflight mechanism for determining network *connectivity*. You determine network connectivity by attempting to connect. If the connection fails, consult the `SCNetworkReachability` API to help diagnose the cause of the failure.

Whether the requests are user-generated and background), the `SCNetworkReachability` API provides a good way to watch for interface availability changes that may require you to reconnect existing connections. When the network interface you are using goes away, you should quickly reconnect to avoid unnecessary delays for the user.

Also, on iOS, if you are connected over a cellular connection, you should quickly reconnect in the background whenever Wi-Fi service becomes available again. Wi-Fi connections use less battery power, are usually faster, and often cost the user less money than cellular connections.

Design for Variable Network Speed

Your program must be prepared for the speed of the network to change, even when the current network interface remains unchanged. For example, when a mobile device user changes locations, performance on Wi-Fi or cellular networks can change significantly, either because of increased interference or because the device was handed off to a busier cell site. It doesn't take a big change in location either; even walking from one room to another can cause significant changes in both Wi-Fi and cellular service speeds.

Further, even ignoring contention and interference, the interface itself tells you nothing about the actual bandwidth available a few hops away. The Wi-Fi network might be fast when the user tries to connect to Google, but the route between the user and *your* server could be going through a cellular modem or a satellite uplink truck. Similarly, a user might have a gigabit Ethernet connection to servers on the local area network, but only a 128-kilobit upstream connection to the outside world. For this reason, you should not make any assumptions about the speed of the network based on the current network interface.

There is only one way to determine the network's speed: use it. After you download a small amount of data, you can establish an initial estimate of the network speed. You should continue to monitor your download rate to maintain an accurate estimate, and then adjust your expectations accordingly. For example, if you are streaming video and you determine that your streaming rate is no longer keeping up with playback, you might switch silently to a lower bandwidth stream on the fly and continue playback as though nothing happened. If you later determine that download speeds have improved, you can switch back just as silently.

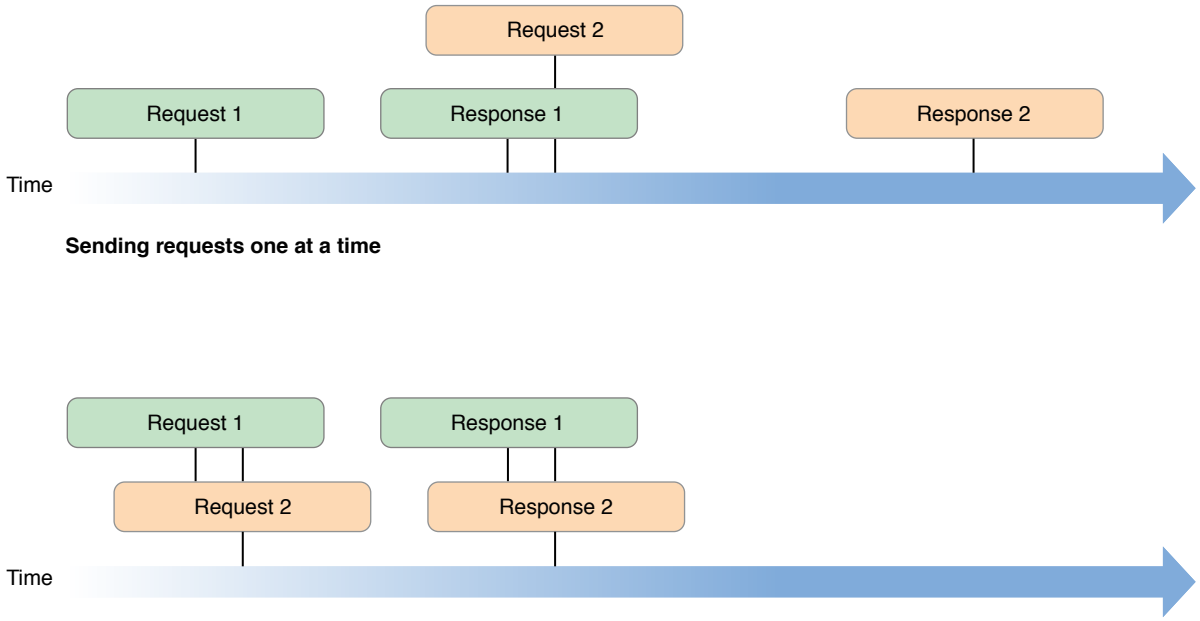
Design for High Latency

As a developer, assume that your users might use a high-latency connection. High latency is particularly common on some types of cellular network interfaces because of the limited number of time slots that can be used by a given device. For example, the round-trip latency over an EDGE connection is often measured in seconds. However, even the half-second latency caused by a satellite connection or a moderately busy DSL connection can cause serious problems if you do not plan for it in your software design.

When an app makes multiple requests to one or more remote hosts, if it waits for the first request to return a result before making the second one, the connection latency becomes additive; the second request is penalized by the latency of the first request in addition to its own, the third request is penalized by the latency of the first two requests, and so on.

To avoid this problem, whenever your program needs to send multiple messages (resource requests, acknowledgments, and so on) that are not dependent on one another, send them all simultaneously rather than waiting for a response to one message before sending the next. Figure 1-1 illustrates the speedup your program gets from sending multiple messages simultaneously.

Figure 1-1 Comparison of response times for simultaneous and sequential requests



If you use `NSURLConnection` in your iOS app, you can easily get a speedup by enabling HTTP pipelining. When pipelining is enabled, your connection automatically sends multiple HTTP requests simultaneously. Enable pipelining by calling the `setHTTPShouldUsePipelining:` method on the `NSMutableURLRequest` object you provide to your connection.

> **Note:** Some servers do not support pipelining. If you connect to a server that does not support pipelining, the connection works but it does not improve performance.

Test Under Various Conditions

Xcode provides a tool called Network Link Conditioner that can simulate various network conditions, including reduced bandwidth, high latency, DNS delays, packet loss, and so on. Before you ship any software that uses networking, you should install this tool, enable it, then run your software to see how it performs under real-world conditions.

Here are a few things to test:

- Make sure your software remains usable even with lousy bandwidth. Tune your bandwidth consumption as much as you can.
- Increase the latency to three or four seconds. Make sure that any user-initiated operation is delayed by only a few seconds, not by a few minutes.
- When the network connection drops packets, your software should continue to function, just more slowly.

You may also find it helpful to use third-party tools such as tcptrace to visualize your software's network access patterns under abusive network conditions.

Assessing Your Networking Needs

Before you can choose a networking API, you need to know a little bit about the networking API families that OS X and iOS offer.

OS X and iOS provide three main user-space networking API layers. The first two—Foundation and CFNetwork (based on Core Foundation)—are frameworks specific to OS X and iOS. The lowest layer, POSIX, is the same as on any other UNIX- or Linux-based operating system.

Within each layer, there are functions or classes that support common networking tasks, such as connecting to remote hosts (protocol streams), downloading the contents of URLs, and discovering services on your local network. These layers are shown in Table 2-1.

Table 2-1 The layers and families of OS X and iOS networking APIs

Layer	Protocol streams	URL loading	Service discovery
Foundation layer	`NSStream`	`NSURLConnection` and `NSURLRequest`	`NSNetService`
Core Foundation layer	`CFStream`	`CFHTTPMessage`	`CFNetService`
POSIX layer	`kqueue`	`libcurl` (for example. Note that this is a third-party API)	DNS Service Discovery
	Built on top of BSD sockets (directly or indirectly)		

You can easily accomplish most client-side networking tasks using only Foundation classes. If you are writing server code or you have specialized needs, you may want to use lower-level frameworks instead. However, as a general rule, you should always choose the highest-level API that meets your needs.

Common Networking Tasks

Before you can decide which specific API to use, you must first assess what networking tasks your program needs to perform.

Support peer-to-peer networking for games. In iOS, the Game Kit framework provides support for peer-to-peer communication, both globally (over the Internet) and locally (using a Bluetooth personal area network or a Wi-Fi LAN).

You can use Game Kit in your app to simplify the following peer-to-peer networking tasks:

- Providing network communication for a multiplayer game
- Providing voice communication

Any peer-to-peer communication that is not covered by the tasks above should be accomplished with lower level networking APIs described later in this section.

To learn more about Game Kit, read *Game Center Programming Guide*.

Support peer-to-peer networking for other apps. In iOS, the Multipeer Connectivity framework provides support for peer-to-peer communication over infrastructure Wi-Fi, peer-to-peer Wi-Fi, and Bluetooth. To learn more, read Discovering and Advertising Network Services (page 24).

Connect to a web server. The preferred way to send and receive short pieces of information is over a standard protocol such as HTTP or HTTPS. By using these existing protocols, you minimize the amount of work needed to support the connection on both the client and server side. HTTP also makes it easy to move to a secure (HTTPS) connection—you just add a certificate on your server and then add a single letter to the first part of the URL.

To learn more about APIs for making HTTP and HTTPS requests, read Making HTTP and HTTPS Requests (page 31).

To learn how to display a web page in your application, read Displaying Web and Multimedia Content (page 29).

Connect to an FTP server. Unless you must do so to maintain compatibility with existing servers, the use of FTP is discouraged. FTP is an old protocol with serious limitations and no real security (data and passwords are sent in cleartext).

With that in mind, if you just need to download a file over FTP, you should use the `NSURLConnection` API and pass it the appropriate URL. This API is described in Making HTTP and HTTPS Requests (page 31), but can also be used with `ftp://` URLs.

For more complex requests, the CFNetwork framework (based on Core Foundation) provides the `CFFTPStream` API for communicating with FTP servers. CFNetwork also provides the `CFURLAccess` API, which can be used for deleting files on an FTP server. The details of these APIs are outside the scope of this document. To learn more, read *CFNetwork Programming Guide*.

Discover and advertise network services. OS X and iOS provide support for DNS Service Discovery, which allows you to describe what services your program provides and discover other services on the user's machine, on nearby machines, or on one of the user's home machines using Back to My Mac. You can then use that information to communicate with other copies of your program or other programs that your program knows how to talk to. For example, OS X uses DNS Service Discovery to let users find nearby printers, stream music in iTunes from nearby computers, share screens in Finder, and so on.

To learn how to discover services on your local area network or on remote servers using DNS Service Discovery, read Discovering and Advertising Network Services (page 24).

Resolve DNS hostnames. OS X and iOS provide Core-Foundation-layer and POSIX-layer name resolver APIs for obtaining IP addresses for a hostname. To learn about these APIs, read Designing for Real-World Networks (page 13). However, if you are resolving hosts because you want to connect to them, you should generally connect by name instead. Read Avoid Resolving DNS Names Before Connecting to a Host (page 54) in Avoiding Common Networking Mistakes (page 49) to learn why.

Use sockets or socket streams. If you need to make network requests in ways that are not supported by higher-level APIs, you can use sockets (at both the POSIX layer and the Core Foundation layer) or socket streams (at the Core Foundation layer). For more information, read Using Sockets and Socket Streams (page 39).

Communicate securely. OS X and iOS support the Transport Layer Security (TLS) protocol and its predecessor, the Secure Sockets Layer (SSL), for encrypted communication and server trust determination. To learn more, read Using Networking Securely (page 41).

Next Steps

Now that you have decided what you want to do, you can easily accomplish a wide array of networking tasks in OS X and iOS, with little or no configuration. Many of the most common networking tasks and the recommended methods for accomplishing them are briefly described in the chapters that follow. Keep in mind that these are not comprehensive API discussions; each chapter ends with links to other documents that provide in-depth information about these tasks.

Discovering and Advertising Network Services

OS X and iOS provide four APIs for discovering and advertising network services:

- `NSNetService`—A high-level Objective-C API suitable for most app developers.
- `CFNetService`—A high-level C API suitable for use in Core Foundation code.
- DNS Service Discovery—A low-level C API suitable for cross-platform code. This API also offers more flexibility than the higher-level APIs.
- Game Kit framework—A high-level Objective-C API that provides peer-to-peer communication support for games, both locally (using infrastructure Wi-Fi and Bluetooth) and globally over the Internet.

In addition to these APIs, iOS offers the Multipeer Connectivity Framework, which provides support for discovering and communicating with instances of your app and related apps on nearby devices using infrastructure Wi-Fi, peer-to-peer Wi-Fi, and Bluetooth.

As a rule, you should use Game Kit only for game-related peer-to-peer networking. For other peer-to-peer networking between iOS devices running iOS 7 and later, you should consider using the Multipeer Connectivity framework.

For compatibility with older versions of iOS, you can also write your own networking code and use `CFNetService` or `NSNetService` to advertise its availability.

> **Note:** On devices that support Bluetooth, Bluetooth communication is automatically used by Game Kit. (Bonjour over Bluetooth can also be enabled when using the DNS Service Discovery C API by setting the `interfaceIndex` to `kDNSServiceFlagsIncludeP2P`. See *Bonjour over Bluetooth on iOS 5 and Later* for details.)

Bonjour Service Overview

A Bonjour service advertisement consists of three parts:

- Service name—This name must be unique to a particular instance of your program running on a particular computer.
- Service type—This must be the same for all instances of your program, and should be registered with the Internet Assigned Numbers Authority (IANA).

- Domain—If the domain value is empty, the host chooses the appropriate domains in which to publish or browse.

When an app browses for Bonjour services, it asks for services matching a particular type in a particular domain, and it gets back a list of matching service names. It should then present an appropriate UI to the user. When the user tells the app to connect to a particular service, the app should then connect to the service using a connect-to-service API. (If this is not possible for some reason, the app can pass the service's hostname and port to a connect-by-name API or, if a connect-by-name API is not available, the app can ask Bonjour to resolve the hostname, and the app can then connect by IP address and port number.)

Publishing a Network Service

Bonjour zero-configuration networking lets you advertise network services, such as a printer or a document syncing service, on a network. There are three ways to publish a network service:

- For Objective-C and Core Foundation code, the recommended way is with the `CFNetServices` API.
- For portable C code that must run on operating systems other than iOS and OS X, the DNS Service Discovery C API is recommended.

You can publish a network service with the following steps:

1. Create a socket to listen for connections to the service. See Writing a TCP-Based Server in *Networking Programming Topics* for the recommended way to listen for connections on a network socket.
2. Create a service object, providing the port of your socket, the domain (usually an empty string), and the service type string of your choosing:
    - With Foundation, initialize an `NSNetService` object with the `initWithDomain:type:name:port:` method.
    - With Core Foundation, create a `CFNetServiceRef` object with the `CFNetServiceCreate` function.
    - With the DNS Service Discovery API, call `DNSServiceRegister` to return a `DNSServiceRef` object.
3. Assign a delegate or callback:
    - With Foundation, assign a delegate to the `NSNetService` object with the `delegate` method.
    - With Core Foundation, assign a client callback to the `CFNetServiceRef` object with the `CFNetServiceSetClient` function.
    - With the DNS Service Discovery API, you should pass a client callback (and, optionally, a pointer to a context object of your choosing) in your call to `DNSServiceRegister`. At this point, you are done except for handling callbacks when they occur.
4. Schedule or reschedule the service, if necessary:

- With Foundation, the service is automatically scheduled on the current run loop in the default mode. If you need to schedule the object on another run loop or in a different mode, you should unschedule it and reschedule it at this point.
- With Core Foundation, you *must* schedule the `CFNetServicesRef` object on a run loop by calling `CFNetServiceScheduleWithRunLoop`.
- With the DNS Service Discovery API, call `DNSServiceSetDispatchQueue` to schedule the service on a dispatch queue. (If you must support an OS prior to OS X v10.7, see the *SRVResolver* sample code project for an example of how to use DNS Service Discovery without Grand Central Dispatch.)

5. Publish the service, if necessary:
   - With Foundation, publish the service by calling the `publish` method.
   - With Core Foundation, publish the service by calling `CFNetServiceRegisterWithOptions`.
   - With the DNS Service Discovery API, no further action is necessary; the service was already published when you called `DNSServiceRegister`.

After your service is published, you can listen for connections on your socket and set up input and output streams when a connection is made.

> **Important:** If you create a custom protocol, you should use a custom service type, and register that service type with IANA. For details, see RFC 6335.

Browsing for and Connecting to a Network Service

The process for finding and resolving a network service is as simple as the process for publishing one. To browse for network services in Objective-C, create an instance of the `NSNetServiceBrowser` class and assign it a delegate. Then, call the `searchForServicesOfType:inDomain:` method on the service browser. The `netServiceBrowser:didFindService:moreComing:` delegate method is called once for every service found.

To connect to a service, first stop the browsing by calling `stop` (unless you have a specific reason to keep browsing), then call the `getInputStream:outputStream:` method on the `NSNetService` object that represents the service. The address of the service is resolved automatically.

You can also use the `CFStreamCreatePairWithSocketToNetService` function with a `CFNetServiceRef` object to connect to a Bonjour service.

Important: If you are using ARC, you should read *NSNetService and Automatic Reference Counting (ARC)*.

Resolving a Network Service

You may need to resolve a network service manually to provide the service's address to an API that does not accept network service names. To resolve a network service in Objective-C, first stop the browsing by calling `stop` (unless you have a specific reason to keep browsing), then call the `resolveWithTimeout:` method on the `NSNetService` object that represents the service.

The `netServiceDidResolveAddress:` method is called on the service's delegate when the service's address has been resolved. You can then access the service's hostname with the `hostName` method or its address information with the `addresses` method. To avoid unnecessary network traffic, you should also call `stop` on the `NSNetService` object as soon as it returns a set of addresses.

Important: The resolution process returns both numerical IP addresses and a hostname. The IP addresses can be an arbitrary mix of IPv4 and IPv6 addresses. Unless you are doing something unusual, you should normally pass the hostname to any API that supports hostnames rather than using the IP addresses directly, because otherwise you would otherwise have to write your own code to try connecting to each of the multiple IP addresses in parallel or in series (described further in Avoid Resolving DNS Names Before Connecting to a Host (page 54)).

The resolver caches the mapping from hostname to IP addresses, so future lookups do not result in any additional network traffic.

Multipeer Connectivity Overview

The Multipeer Connectivity Framework provides a layer on top of Bonjour that lets you communicate with apps running on nearby devices (over infrastructure Wi-Fi, peer-to-peer Wi-Fi, and Bluetooth) without having to write lots of networking code specific to your app.

With Multipeer Connectivity, your app advertises its availability. It can then discover other instances of your app (or other apps that share the same service type) running on nearby devices, and can invite those nearby peers to join a session. If they accept the invitation, your app can send messages and files to one or more of the connected peers with just a single method call.

Important: As with Bonjour, your app must provide a service type, and you should register that service type with IANA. For details, see RFC 6335.

If you need stream-based communication, your app can open a unidirectional stream to any connected peer (which can also open a unidirectional stream back to your app in response).

Finally, Multipeer Connectivity provides the ability to share small amounts of data (such as the user's screen name) outside the context of a session, if desired, allowing you to provide the user with information that he or she can use when choosing peers to invite into a session.

To Learn More

Multipeer Connectivity—Read *Multipeer Connectivity Framework Reference* and the *MultipeerGroupChat* sample code project.

Game Kit—Read *Game Center Programming Guide*, *GameKit Framework Reference*, and the *GKRocket* and *GKTank* sample code projects.

`NSNetService`—Read *NSNetServices and CFNetServices Programming Guide*, *NSNetServiceBrowser Class Reference*, *NSNetServiceBrowserDelegate Protocol Reference*, and *NSNetServiceDelegate Protocol Reference*. For sample code, see the *RemoteCurrency* sample code project.

`CFNetService`—Read *NSNetServices and CFNetServices Programming Guide* and *CFNetServices Reference*.

DNS Service Discovery—Read *DNS Service Discovery Programming Guide* and *DNS Service Discovery C Reference*.

Displaying Web and Multimedia Content

OS X and iOS provide an assortment of APIs to allow you to display web content and streaming multimedia content. In general, if these higher-level multimedia- and web-specific APIs meet your needs, you should use them rather than using networking APIs directly. The sections below briefly summarize these APIs.

Opening Web Content or Streaming Media in the Default Application

To open a webpage or streaming URL in the user's default browser or media viewer:

- In iOS, use the `openURL:` method of the `UIApplication` class.

  For a real-world example, see QA1629: *Launching the App Store from an iOS application*.

- In OS X, use the `LSOpenCFURLRef` or `LSOpenFromURLSpec` functions in the Launch Services API.

  For details, see Launch Services Tasks in *Launch Services Programming Guide*.

Displaying Web Content in Your Application

OS X and iOS provide an easy way to load and display a webpage with the WebKit engine, the same rendering engine used by Safari.

- In OS X, you load web content with the `WebView` class. You can add a web view by including it in your application's nib file or by programmatically constructing a `WebView` object and calling the `initWithFrame:frameName:groupName:` method. Load content by calling the `loadRequest:` method on the web view's main frame (which you can obtain with the `mainFrame` method).
- In iOS, you load web content with the `loadRequest:` method of the `UIWebView` class. You can add a web view by including it in your application's nib file or by programmatically creating a `UIWebView` object and initializing it with the `initWithFrame:` method.

> **Note:** Web views in iOS don't provide access to their underlying connection when they load data, which means a connection that can't be resolved automatically (such as a connection that requires authentication) fails.

For more information, see Simple Browsing in *WebKit Objective-C Programming Guide* (OS X) and *UIWebView Class Reference* (iOS).

Displaying Streaming Multimedia Content in Your Application

There are several frameworks available for displaying streaming multimedia content in OS X and iOS:

- In OS X, use the QTKit Framework for basic playback or the AV Foundation framework for more complex functionality.
- In iOS, use the Media Player Framework for basic playback or the AV Foundation framework for more complex functionality.

For more information, read *Getting Started with Audio & Video*, *Multimedia Programming Guide* (iOS), *QTKit Application Programming Guide* (OS X), and *AVFoundation Programming Guide*.

Making HTTP and HTTPS Requests

OS X and iOS provide a number of general-purpose APIs for making HTTP and HTTPS requests. With these APIs, you can download files to disk, make simple HTTP and HTTPS requests, or precisely tune your request to the specific requirements of your server infrastructure.

When choosing an API, you should first consider why you are making an HTTP request:

- If you are writing a Newsstand app, you should use the `NKAssetDownload` API to download content in the background.
- If you need to download a file to disk in OS X, the easiest way is to use the `NSURLDownload` class. For details, see Downloading the Contents of a URL to Disk (page 33).
- You should use `CFHTTPStream` if any of the following are true:
  - You have a strict requirement not to use Objective-C.
  - You need to override proxy settings.
  - You need to be compatible with a particular non-compliant server.

  For more information, see Making Requests Using Core Foundation (page 37).
- Otherwise, you should generally use the `NSURLSession` or `NSURLConnection` APIs.

The sections below describe these APIs in more detail.

> **Note:** If you have specific needs, you can also write your own HTTP client implementation using socket or socket-stream APIs. These APIs are described in Using Sockets and Socket Streams (page 39).

Making Requests Using Foundation

The following tasks describe common operations with the `NSURLSession` class, the `NSURLConnection` class, and related classes.

Retrieving the Contents of a URL without Delegates

If you just need to retrieve the contents of a URL and do something with the results at the end, in OS X v10.9 and later or iOS 7 and later, you should use the `NSURLSession` class. You can also use the `NSURLConnection` class for compatibility with earlier versions of OS X and iOS.

To do this, call one of the following methods: `dataTaskWithRequest:completionHandler:` (`NSURLSession`), `dataTaskWithURL:completionHandler:` (`NSURLSession`), or `sendAsynchronousRequest:queue:completionHandler:` (`NSURLConnection`). Your app must provide the following information:

- As appropriate, either an `NSURL` object or a filled-out `NSURLRequest` object that provides the URL, body data, and any other information that might be required for your particular request.
- A completion handler block that runs whenever the transfer finishes or fails.
- For `NSURLConnection`, an `NSOperation` queue on which your block should run.

If the transfer succeeds, the contents of the request are passed to the callback handler block as an `NSData` object and an `NSURLResponse` object for the request. If the URL loading system is unable to retrieve the contents of the URL, an `NSError` object is passed as the third parameter.

Retrieving the Contents of a URL with Delegates

If your app needs more control over your request, such as controlling whether redirects are followed, performing custom authentication, or obtaining the data piecewise as it is received, you can use the `NSURLSession` class with a custom delegate. For compatibility with earlier versions of OS X and iOS, you can also use the `NSURLConnection` class with a custom delegate.

For the most part, the `NSURLSession` and `NSURLConnection` classes work similarly at a high level. However, there are a few significant differences:

- The `NSURLSession` API provides support for download tasks that behave much like the `NSURLDownload` class. This usage is described further in Downloading the Contents of a URL to Disk (page 33).
- When you create an `NSURLSession` object, you provide a reusable configuration object that encapsulates many common configuration options. With `NSURLConnection`, you must set those options on each connection independently.
- An `NSURLConnection` object handles a single request and any follow-on requests.

  An `NSURLSession` object manages multiple tasks, each of which represents a single URL request and any follow-on requests. You usually create a session when your app launches, then create tasks in much the same way that you would create `NSURLConnection` objects.

- With `NSURLConnection`, each connection object has a separate delegate. With `NSURLSession`, the delegate is shared across all tasks within a session. If you need to use a different delegate, you must create a new session.

When you initialize an `NSURLSession` or `NSURLConnection` object, the connection or session is automatically scheduled in the current run loop in the default run loop mode.

The delegate you provide receives notifications throughout the connection process, including intermittent calls to the `URLSession:dataTask:didReceiveData:` or `connection:didReceiveData:` method when a connection receives additional data from the server. It is the delegate's responsibility to keep track of the data it has already received, if necessary. As a rule:

- If the data can be processed a piece at a time, do so. For example, you might use a streaming XML parser.
- If the data is small, you might append it to an `NSMutableData` object.
- If the data is large, you should write it to a file and process it upon completion of the transfer.

When the `URLSession:task:didCompleteWithError:` or `connectionDidFinishLoading:` method is called, the delegate has received the entirety of the URL's contents.

Downloading the Contents of a URL to Disk

In OS X v10.9 and later or iOS 7 and later, if you need to download a URL and store the results as a file, but do not need to process the data in flight, the `NSURLSession` class lets you download the URL directly to a file on disk in a single step (as opposed to loading the URL into memory and then writing it out yourself). The `NSURLSession` class also allows you to pause and resume downloads, restart failed downloads, and continue downloading while the app is suspended, crashed, or otherwise not running.

In iOS, the `NSURLSession` class also launches your app in the background whenever a download finishes so that you can perform any app-specific processing on the file.

> **Note:** In older versions of OS X, you can also download files to disk with the `NSURLDownload` class. The `NSURLDownload` class does not provide the ability to download files while the app is not running.
>
> In older versions of iOS, you must use an `NSURLConnection` object to download the data to memory, then write the data to a file yourself.

To use the `NSURLSession` class for downloading, your code must do the following:

1. Create a session with a custom delegate and the configuration object of your choice:
   - If you want downloads to continue while your app is not running, you must provide a background session configuration object (with a unique identifier) when you create the session.

- If you do not care about background downloading, you can create the session using any of the provided session configuration object types.

2. Create and resume one or more download tasks within the session.
3. Wait until your delegate receives calls from the task or session. In particular, you must implement the `URLSession:downloadTask:didFinishDownloadingToURL:` method to do something with a file when the download finishes and the `URLSession:task:didCompleteWithError:` call to handle any errors.

Note: The above steps are a greatly simplified view; depending on your needs, you may wish for your session delegate to handle a number of other delegate methods for custom authentication, redirect handling, and so on.

Making a POST Request

You can make an HTTP or HTTPS POST request in nearly the same way you would make any other URL request (described in Retrieving the Contents of a URL with Delegates (page 32)). The main difference is that you must first configure the `NSMutableURLRequest` object you provide to the `initWithRequest:delegate:` method.

You also need to construct the body data. You can do this in one of three ways:

- For uploading short, in-memory data, you should URL-encode an existing piece of data. This process is described in Encoding URL Data.
- For uploading file data from disk, call the `setHTTPBodyStream:` method to tell `NSMutableURLRequest` to read from an `NSInputStream` and use the resulting data as the body content.
- For large blocks of constructed data, call `CFStreamCreateBoundPair` to create a pair of streams, then call the `setHTTPBodyStream:` method to tell `NSMutableURLRequest` to use one of those streams as the source for its body content. By writing into the other stream, you can send the data a piece at a time.

  Depending on how you handle things on the server side, you may also want to URL-encode the data you send.)

To specify a different content type for the request, use the `setValue:forHTTPHeaderField:` method. If you do, make sure your body data is properly formatted for that content type.

To obtain a progress estimate for a POST request, implement a `connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:` method in the connection's delegate.

Configuring Authentication

Performing authentication with `NSURLSession` and `NSURLConnection` is relatively straightforward. The way you do this depends on the class you use and on the version of OS X or iOS that you are targeting.

For the `NSURLSession` class, your delegate should implement the `URLSession:task:didReceiveChallenge:completionHandler:` method. In this method, you perform whatever operations are needed to determine how to respond to the challenge, then call the provided completion handler with a constant that indicates how the URL Loading System should proceed and, optionally, a credential to use for authentication purposes.

For the `NSURLConnection` class:

- In OS X v10.7 and newer or iOS 5 and newer, your delegate should implement the `connection:willSendRequestForAuthenticationChallenge:` method. This method must call a method on the sender (the `NSURLConnection` object) to tell it how to proceed.
- In earlier versions, your delegate should implement both the `connection:canAuthenticateAgainstProtectionSpace:` and `connection:didReceiveAuthenticationChallenge:` methods.

  The `connection:didReceiveAuthenticationChallenge:` method is equivalent to the `connection:willSendRequestForAuthenticationChallenge:` method in later versions, and calls a method on the sender (the `NSURLConnection` object) to tell it how to proceed.

  The `connection:canAuthenticateAgainstProtectionSpace:` method should return `YES` if `[protectionSpace authenticationMethod]` is any of `NSURLAuthenticationMethodDefault`, `NSURLAuthenticationMethodHTTPBasic`, `NSURLAuthenticationMethodHTTPDigest`, `NSURLAuthenticationMethodHTMLForm`, `NSURLAuthenticationMethodNegotiate`, or `NSURLAuthenticationMethodNTLM`.

Possible Responses to an Authentication Challenge

Regardless of which class you use, your authentication handler method must examine the authentication challenge and tell the URL Loading System how to proceed:

- To provide a credential for authentication, pass `NSURLSessionAuthChallengeUseCredential` as the disposition (for `NSURLSession`) or call `useCredential:forAuthenticationChallenge:` (for `NSURLConnection`).

  For information about creating a credential object, read Creating a Credential Object (page 36).
- To continue the request without providing authentication, pass `NSURLSessionAuthChallengeUseCredential` as the disposition with a `nil` credential (for `NSURLSession`) or call `continueWithoutCredentialForAuthenticationChallenge:` (for `NSURLConnection`).

- To cancel the authentication challenge, pass `NSURLSessionAuthChallengeCancelAuthenticationChallenge` as the disposition (for `NSURLSession`) or call `cancelAuthenticationChallenge:` (for `NSURLConnection`). If you cancel the authentication challenge, the stream delegate's error method is called.
- To tell the operating system to handle the challenge as it ordinarily would, pass `NSURLSessionAuthChallengePerformDefaultHandling` as the disposition (for `NSURLSession`) or call `performDefaultHandlingForAuthenticationChallenge:` (for `NSURLConnection`). If you request default handling, then the operating system sends any appropriate credentials that exist in the credentials cache.

  > **Note:** The `performDefaultHandlingForAuthenticationChallenge:` method was not supported prior to OS X v10.7 or iOS 5.

- To reject a particular type of authentication during the negotiation process, with the intent to accept a different method, pass `NSURLSessionAuthChallengeRejectProtectionSpace` as the disposition (for `NSURLSession`) or call `rejectProtectionSpaceAndContinueWithChallenge:` (for `NSURLConnection`).

  > **Note:** The `rejectProtectionSpaceAndContinueWithChallenge:` method was not supported prior to OS X v10.7 or iOS 5.

Creating a Credential Object

Within your delegate's `connection:willSendRequestForAuthenticationChallenge:` or `connection:didReceiveAuthenticationChallenge:` method, you may need to provide an `NSURLCredential` object that provides the actual authentication information.

- For simple login/password authentication, call `credentialWithUser:password:persistence:`.
- For certificate-based authentication, call `credentialWithIdentity:certificates:persistence:` with a `SecIdentityRef` object (which is usually obtained from the user's keychain by calling `SecItemCopyMatching`).

Further Information

To learn more about the `NSURLSession` API, read *URL Session Programming Guide*. For related sample code, see *SimpleURLConnections*, *AdvancedURLConnections*, and *Using NSXMLParser to parse XML documents*.

For details about the `NSURLConnection` API, read *URL Session Programming Guide*.

To learn more about using the `NSStream` API for making HTTP requests, read Setting Up Socket Streams in *Stream Programming Guide*.

For an example of the `setHTTPBodyStream:` method and the `CFStreamCreateBoundPair` function, see *SimpleURLConnections* in the iOS library. (The sample as a whole is designed to build and run on iOS, but the networking portions of the code are also useful on OS X.)

Making Requests Using Core Foundation

Other than the syntax details, the request functionality in Core Foundation is closely related to what is available at the Foundation layer. Thus, the examples in Making Requests Using Foundation (page 31) should be helpful in understanding how to make requests using the `CFHTTPStream` API.

The Core Foundation URL Access Utilities are a C-language API that is part of the Core Foundation framework. To learn more, read *Core Foundation URL Access Utilities Reference*.

The `CFHTTPStream` API is a C-language API that is part of the Core Foundation framework. (You can, of course, use it in Objective-C code.) To learn more, read Communicating with HTTP Servers and Communicating with Authenticating HTTP Servers in *CFNetwork Programming Guide*.

These APIs are the most flexible way to communicate with an HTTP server (short of using sockets or socket streams directly), providing complete control over the message body as sent to the remote server, and control over most of the message headers as well. These APIs are also more complex, and thus should be used only if higher-level APIs cannot support your needs—for example, if you need to override the default system proxies.

Working with Web Services

If you are incorporating client-side web services communication in your OS X program, you can take advantage of a number of technologies:

- The `NSJSONSerialization` class converts between native Cocoa objects and JavaScript Object Notation (JSON).
- The `NSXMLParser` class provides a Cocoa API for SAX-style (streaming) parsing of XML content.
- The libxml2 library provides a cross-platform C API for SAX-style (streaming) and DOM-style (tree-based) parsing of XML content. For libxml2 documentation, see http://xmlsoft.org/.
- The `NSXMLDocument` API (in OS X only) provides DOM-style support for XML content.

In addition, a number of third-party libraries exist for working with web services.

Important: The Web Services Core framework is deprecated and should not be used for new development.

Using Sockets and Socket Streams

This chapter describes ways to make socket connections that are completely under the control of your program. Most programs would be better served by higher-level APIs such as `NSURLConnection`, which was described in previous chapters. These APIs should be used only if you need to support some protocol other than the protocols supported by built-in Cocoa or Core Foundation functionality.

Choosing a Socket API

At almost every level of networking, software can be divided into two categories: clients (programs that connect to other apps) and services (programs that other apps connect to). At a high level, these lines are clear. Most programs written using high-level APIs are purely clients. At a lower level, however, the lines are often blurry.

Socket and stream programming generally falls into one of the following broad categories:

- Packet-based communication—Programs that operate on one packet at a time, listening for incoming packets, then sending packets in reply.

  With packet-based communication, the only differences between clients and servers are the contents of the packets that each program sends and receives, and (presumably) what each program does with the data. The networking code itself is identical.

- Stream-based clients—Programs that use TCP to send and receive data as two continuous streams of bytes, one in each direction.

  With stream-based communication, clients and servers are somewhat more distinct. The actual data handling part of clients and servers is similar, but the way that the program initially constructs the communication channel is very different.

The API you choose for socket-based connections depends on whether you are making a connection to another host or receiving a connection from another host. It also depends on whether you are using TCP or some other protocol. Here are a few factors to consider:

- In OS X, if you already have networking code that is shared with non-Apple platforms, you can use POSIX C networking APIs and continue to use your networking code as-is (on a separate thread). If your program is based on a Core Foundation or Cocoa (Foundation) run loop, you can also use the Core Foundation `CFStream` API to integrate the POSIX networking code into your overall architecture on the main thread. Alternatively, if you are using Grand Central Dispatch (GCD), you can add a socket as a dispatch source.

In iOS, POSIX networking is discouraged because it does not activate the cellular radio or on-demand VPN. Thus, as a general rule, you should separate the networking code from any common data processing functionality and rewrite the networking code using higher-level APIs.

> **Note:** If you use POSIX networking code, you should be aware that the POSIX networking API is not protocol-agnostic (you must handle some of the differences between IPv4 and IPv6 yourself). It is a connect-by-IP API rather than a connect-by-name API, which means that you must do a lot of extra work if you want to achieve the same initial-connection performance and robustness that higher-level APIs give you for free. Before you decide to reuse existing POSIX networking code, be sure to read Avoid Resolving DNS Names Before Connecting to a Host (page 54) in Avoiding Common Networking Mistakes (page 49).

- For daemons and services that listen on a port, or for non-TCP connections, use POSIX or Core Foundation (`CFSocket`) C networking APIs.
- For client code in Objective-C, use Foundation Objective-C networking APIs. Foundation defines high-level classes for managing URL connections, socket streams, network services, and other networking tasks. It is also the primary non-UI Objective-C framework in OS X and iOS, providing routines for run loops, string handling, collection objects, file access, and so on.
- For client code in C, use Core Foundation C networking APIs—part of the CFNetwork framework. The Core Foundation framework and the CFNetwork framework are two of the primary C-language frameworks in OS X and iOS. Together they define the functions and structures upon which the Foundation networking classes are built.

> **Note:** In older versions of OS X, CFNetwork is a subframework of the Core Services framework.

To Learn More

To learn more about how to use sockets and socket streams, read Using Sockets and Socket Streams.

Using Networking Securely

Whether you are writing a banking app or a game, if your program uses networking, it should be secure. For all but the most trivial pieces of data, it's impossible for software to determine whether a user's data is confidential, embarrassing, or even dangerous. Large numbers of seemingly insignificant pieces of information can, in aggregate, be a much greater concern than the sum of its parts.

For these reasons, always assume that every piece of data your program encounters could contain a bank account number or a password, and secure it accordingly.

Some attacks your program might encounter include:

- Snooping—Attacks in which a third party sniffs your program's data in transit.
- Man-in-the-middle attacks—Attacks in which a third party interposes its own computer between your program and a server. Man-in-the-middle attacks include:

  Spoofing and phishing—Creating false servers that masquerade as legitimate servers.

  Tampering—Modifying data between the server and your program.

  Session hijacking—Capturing authentication information and using it to pose as your users.

- Injection attacks—Attacks in which specially crafted data can cause client or server software to execute commands other than the inspected ones. This commonly occurs when the program talks to a script interpreter, such as a shell or a SQL database server.
- Buffer overflows and numeric overflows—Attacks in which specially crafted data can cause a program to read or write data in parts of its address space where it shouldn't, potentially leading to execution of arbitrary executable code, disclosure of private information, or both.

This chapter explains how to defend against snooping and man-in-the-middle attacks. To learn more about injection attacks, buffer overflows, and other aspects of software security, read *Secure Coding Guide*.

Enabling TLS or SSL

The Transport Layer Security (TLS) protocol provides data encryption for socket-based communication, along with authentication of servers and (optionally) clients to prevent spoofing.

OS X and iOS also provide support for the Secure Sockets Layer (SSL) protocol. Because TLS is the successor to SSL, OS X and iOS use TLS by default if both protocols are supported.

> **Note:** TLS and SSL are primarily designed for use in a client-server model. It's more difficult to ensure secure communication in a peer-to-peer environment with these protocols.

Connecting Securely to a URL

Connecting to a URL via TLS is trivial. When you create an `NSURLRequest` object to provide to the `initWithRequest:delegate:` method, specify `https` as the scheme of the URL instead of `http`. The connection uses TLS automatically with no additional configuration.

Connecting Securely Using Streams

You can use TLS with an `NSStream` object by calling `setProperty:forKey:` on it. Specify `NSStreamSocketSecurityLevelNegotiatedSSL` as the `property` parameter and `NSStreamSocketSecurityLevelKey` as the `key` parameter. If you need to work around compatibility bugs, you can also specify a more specific protocol, such as `NSStreamSocketSecurityLevelTLSv1`.

Connecting Securely Using BSD Sockets

When making secure connections, if possible, you should use `NSStream` (as described in the previous section) instead of using sockets directly. However, if you must work with BSD sockets directly, you must perform the SSL or TLS encryption and decryption yourself. Depending on your platform, there are two ways to do this:

- In OS X, or in iOS 5 and later, you can use the Secure Transport API in the Security framework to handle your SSL and TLS handshaking, encryption, and decryption. See *Secure Transport Reference* for details.
- In iOS and OS X, you can download an open source SSL or TLS implementation, such as OpenSSL and include a compiled copy of that library (or some portion thereof) in your app bundle (or alongside your nonbundled program). Be sure to comply with the licensing terms of any third-party libraries you might use.

Note: Although a version of OpenSSL libraries is included as part of OS X, the OpenSSL library does not guarantee binary compatibility across different versions of OpenSSL. For this reason, linking to the built-in copy of OpenSSL is deprecated as of OS X v10.7. If you want to use OpenSSL, provide your own copy of the library so that you can control precisely which version of OpenSSL your program is linked against.

Using Other Security Protocols in OS X

In addition to the default Secure Transport implementation of TLS, the following network security protocols are available in OS X:

- The Kerberos protocol is available via the Kerberos framework. This protocol provides support for single sign-on authentication over a network. For more information, read *Security Overview* and *Authentication, Authorization, and Permissions Guide*.
- The Secure Shell (SSH) protocol is available. This protocol is commonly used for logging in to remote hosts using the Terminal app. See `ssh` for more information.
- The OpenSSL implementation of TLS is available, but the preinstalled OpenSSL library is deprecated in OS X v10.7 and later for binary compatibility reasons. If you require OpenSSL, provide your own copy of this library instead, and statically link it into your program.

Common Mistakes

There are a number of common mistakes developers make when writing secure networking code. This section provides suggestions for avoiding several of these mistakes.

Be Careful Who You Trust

If your app sends or receives potentially confidential data to or from a server, be certain that it authenticates the server to ensure that it has not been spoofed. Be sure your server authenticates the client correctly to avoid providing data to the wrong user. Also, be certain that the connection is established using appropriate encryption.

Similarly, be sure that you store data only when necessary and provide it only to the minimum extent necessary to perform a task. For example, to maximize privacy of users' personal information, you might store your web servers' databases on separate servers, configured to accept SQL queries only from your web servers, and with limited connectivity to the Internet as a whole. In other words, use proper privilege separation.

For more information, read Designing Secure Helpers and Daemons in *Secure Coding Guide*.

Be Careful What Data You Trust

Every program is at risk of attack by someone providing malformed or malicious content. This is particularly true if your program obtains data from untrusted servers, or if your program obtains untrusted data from trusted servers (forum posts, for example).

To protect against such attacks, your program should carefully examine all data received from the network or from disk (because the user might have downloaded that data). If the data appears malformed in any way, do not process it as you would other data.

For more information, read Validating Input and Interprocess Communication in *Secure Coding Guide*.

Know That Many Tiny Leaks Can Add Up to a Flood

Always take steps to ensure that the contents of your app's Internet traffic remains private. Although certain information may seem harmless by itself, a skilled attacker can combine that information with other information to discover trends that might be a far greater cause for concern than any single data point by itself—a process known as *data mining*.

For example, if someone wants to break into your house, a single post from the library on a Saturday evening is probably harmless, but posts from the library at about the same time of day every Saturday for several weeks in a row might not be so harmless, because they indicate your habits.

The data need not even be all about the same thing to cause harm. Knowing that someone likes to watch a particular TV show might be harmless, but a complete profile of the sorts of shows that someone enjoys, the products he or she buys, and the friends he or she interacts with might correlate strongly with some attribute that the person considers private, such as religion or sexual orientation. In one particularly impressive (and possibly apocryphal) story, a retail chain reportedly recognized that a man's daughter was pregnant based on her purchasing decisions even before her father did.

The risk of disclosure in aggregate is particularly problematic when it comes to things like identity theft. Your app might leak a phone number, another app might leak a postal address, and so on. After the attacker has amassed enough information about a victim, he or she can use social engineering techniques to convince a third party to give him or her even more information, resulting in a feedback loop of information gathering with devastating consequences.

For this reason, it's important that your app use encrypted communication at all times, for all connections, unless it is infeasible to do so. You never know when that seemingly harmless piece of information, when combined with another seemingly harmless piece of information, might prove damaging or hurtful.

Install Certificates Correctly

When connecting to a server using TLS or SSL, if your app gets an error saying that the certificate authority that signed your certificate is unknown, assuming that you obtained your certificate from a reputable certificate authority, this almost invariably means your certificate chain is missing or incomplete.

When your server accepts a connection encrypted with TLS or SSL, it provides two things: your server's SSL certificate and a complete chain of SSL certificates, beginning with your server's certificate and ending with a certificate signed by one of the trusted anchor certificates recognized by the operating system. If there are certificates missing in your chain, you'll get this error because the certificates earlier in the chain cannot be verified without the certificates later in the chain.

To see what your server is actually sending out, type the following command in Terminal (replacing `www.example.com` with your actual domain name) and press Return:

```
openssl s_client -showcerts -connect www.example.com:443
```

When you type this command, you should see your server's certificate, followed by a series of intermediate certificates. If you don't, check your server's configuration. To obtain the correct certificates to put in your server's certificate chain file, contact the certificate authority that provided your server's SSL certificate.

Never Disable Certificate Chain Validation (Unless You Validate Them Yourself)

Disabling chain validation eliminates any benefit you might otherwise have gotten from using a secure connection. The resulting connection is no safer than sending the request using unencrypted HTTP because it provides no protection from spoofing by a fake server.

If you are using server certificates from a trusted certificate authority, be sure your certificates are installed correctly (see the previous section).

If you are working with self-signed certificates temporarily, you should add them to your test machines' trusted anchors list. In OS X, you can do this using the Keychain Access utility. In iOS, you can use the `SecTrustCopyAnchorCertificates`, `SecTrustCreateWithCertificates`, and `SecTrustSetAnchorCertificates` functions within your program.

If you need to specifically allow a single self-signed certificate or a certificate signed for a different (specific) host, or if you need to allow a certificate only for a single connection, you can learn safe ways to do this by reading Overriding TLS Chain Validation Correctly.

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.

Avoiding Common Networking Mistakes

When writing networking-based software, developers often make a few common design and usage mistakes that can cause serious performance problems, crashes, and other misbehavior. This chapter highlights a few of those mistakes and describes how to avoid or fix them.

Clean Up Your Connections

TCP connections remain open until either the connection is explicitly closed or a timeout occurs. Unless TCP keepalive is enabled for the connection, a timeout occurs only if there is data waiting to be transmitted that cannot be delivered. This means that if you do not close your idle TCP connections, they will remain open until your program quits.

The recommended way to enable TCP keepalive is by setting the `SO_KEEPALIVE` flag on the socket with `setsockopt`.

> **Note:** In OS X, you can also globally change the behavior of all sockets on a particular machine by setting the `net.inet.tcp.always_keepalive` sysctl to a nonzero value. You should not do this in publicly shipping software because this flag affects the behavior of other software on the system. However, this flag can be useful for diagnosing and working around misbehaving software. See the `sysctl` man page for details.

Avoid POSIX Sockets and CFSocket on iOS Where Possible

Using POSIX sockets directly has both advantages and disadvantages relative to using them through a higher-level API. The main advantages are:

- Sockets greatly simplify porting networking code to and from non-Apple platforms.
- You can support protocols other than TCP.

The main disadvantages are:

- Sockets have many complexities that are handled for you by higher-level APIs. Thus, you will have to write more code, which usually means more bugs.

- In iOS, using sockets directly using POSIX functions or `CFSocket` does not automatically activate the device's cellular modem or on-demand VPN.

The most appropriate times to use sockets directly are when you are developing a cross-platform tool or high-performance server software. In other circumstances, you typically should use a higher-level API.

Avoid Synchronous Networking Calls on the Main Thread

If you are performing network operations on your main thread, you must use *only* asynchronous calls.

Network communication is inherently prone to delays. For example, a DNS request that times out can take upwards of half a minute, and a `connect` can take even longer. If you perform a synchronous networking call—a call that waits for a response and then returns the data—the thread that made the call becomes blocked in the kernel until the operation either completes or fails. If that thread happens to be your program's main thread, your program becomes unresponsive.

In an OS X GUI app, this causes the spinning wait cursor to appear. Menus become unresponsive, clicks and keystrokes are delayed or lost, and your users may become frustrated.

In iOS, your app is killed by the watchdog timer if it doesn't respond to user interface events for a predetermined amount of time. The timeouts for most networking operations are longer than that of the iOS watchdog timer. Thus, your app is *guaranteed* to be killed if your network connection fails during a synchronous networking call.

If your iOS app generates a crash report with the exception code `0x8badfood`, this means that the iOS watchdog timer killed your app because it was unresponsive; such a crash may have been caused by a synchronous networking call.

Cocoa (Foundation) and CFNetwork (Core Foundation) Code

The easiest and most common way to perform networking asynchronously is to schedule your networking object in the current thread's run loop.

All Foundation or `CFNetwork` networking objects—including `NSURLConnection`, `NSStream` / `CFStream`, `NSNetService` / `CFNetService`, and `CFSocket`—have built-in run-loop integration. Each of these objects has a set of delegate methods or callback functions that are called throughout your object's network communication.

If you want to perform a computationally intensive task (such as processing a large downloaded file) over the course of your network communication, you should do so on a separate thread. The `NSOperation` class lets you encapsulate such a task in an operation object, which you can easily run on a separate thread by adding it to an `NSOperationQueue` object.

For more information, read Run Loops in *Threading Programming Guide*. For sample code, see *LinkedImageFetcher* and *ListAdder*.

POSIX Code

POSIX networking presents some unique challenges, and thus has some unique tips:

Create sockets correctly. The proper call for creating a TCP socket is:

```
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);  // IPv4
socket(PF_INET6, SOCK_STREAM, IPPROTO_TCP); // IPv6
```

> **Note:** A number of socket tutorials incorrectly use `PF_INET` or `AF_INET` for the third parameter, which apparently works on a few operating systems, but fails completely on OS X because the numerical value of `AF_INET` corresponds with that of `IPPROTO_IGMP`, not `IPPROTO_TCP`.
>
> Also, a number of socket tutorials incorrectly use `AF_INET` for the first parameter, which works on most platforms because the constants have the same value, but is not guaranteed to work.

You can create a UDP socket by replacing `IPPROTO_TCP` with `IPPROTO_UDP` in the snippet above and replacing `SOCK_STREAM` with `SOCK_DGRAM`.

If you use the select system call, keep track of which sockets are actually in use. A number of poorly written socket-based programs incorrectly assume that they own every socket or file descriptor from 3 (or 0) up through the highest-numbered socket that they currently have open, and use a simple `for` loop to fill in the file descriptor set. This not only can result in poor performance, but also can cause incorrect behavior if the daemon opens files that may end up with file descriptor numbers in that range.

Instead, you should keep two pieces of state in your networking code:

- An integer that keeps track of the highest-numbered socket that you currently have open. You must update this (if needed) whenever you open or close a new socket.
- A complete `FD_SET` in which the relevant bit is set for every open socket. Instead of passing this set to the `select` system call (which destroys the contents of any descriptor sets passed as parameters), you should copy the descriptor set with `FD_COPY` and pass the copy to `select`.

> **Note:** The `FD_COPY` function in OS X is highly optimized. For maximum performance, do not attempt to copy the file descriptor set yourself using a `for` loop.

Alternatively, if you have an array of file descriptors (or some other way to keep track of them), you can construct a new `FD_SET` from that array.

Use run loops and nonblocking I/O to manage asynchronous reads. In all but the most trivial command-line tools, POSIX networking should always be performed in a run-loop-based fashion using either GCD, the `kqueue` API, or the `select` system call.

Avoid synchronous networking on the main thread. If you are using synchronous calls, POSIX networking should never be performed on the main program thread of any GUI application or other interactive tool. This includes:

- Reading from and writing to sockets that are not set to non-blocking.
- Connecting a socket.
- Performing DNS lookups (particularly with `getaddrinfo`, `getnameinfo`, `getaddrinfo`, and `gethostbyaddr`).

Instead, either perform the synchronous calls on a separate thread or use an API that performs these operations asynchronously, such as GCD, the `kqueue` API, or higher-level APIs that integrate with Cocoa run loops (`CFSocket` or `CFStream`, for example).

For more information about the options available to you, read Assessing Your Networking Needs (page 21) and Using Sockets and Socket Streams (page 39).

Set appropriate timeouts if you are using select. The `select` system call allows you to specify a period of time to wait before returning control to your code. The value you choose for this timeout is very important:

- If your code needs to perform other operations in the background, this timeout value determines how often those other operations run. The shorter the value, the more frequently your code can do other things in the background, but the more CPU cycles it uses while waiting.

  In general, if your app is waking up more than a few times per second (or even once per second on iOS), you should probably be doing that work on a separate thread. Moreover, this is usually a sign that you're doing something wrong, such as polling to see whether a file has been deleted or modified instead of using a more appropriate notification-based technology such as the `kqueue` API.

  If you are using this wakeup to check to see if another thread within your application has done something, you should consider using a pair of connected sockets instead. To do this, first a socket pair with `socketpair`. Then add one of the connected sockets to your descriptor set. When the other thread needs

to tell your networking thread about an event, it can wake your networking thread by writing data into the other socket. Be sure to keep track of which sockets were created in this way so that your code can handle the data differently depending on whether it came from an outside connection or from within your app.

- If your code does not need to perform any operations in the background, you should pass `NULL`. Passing `NULL` ensures that your code takes as little CPU as possible while waiting for incoming connections or data.

In addition, if you are using timeouts, be aware that the `select` system call returns `EINTR` when the timer fires. Your code should be prepared for this return value and should not treat it as an error.

Use POSIX sockets efficiently (if at all). If you are using POSIX sockets directly:

- Handle or disable `SIGPIPE`.

  When a connection closes, by default, your process receives a `SIGPIPE` signal. If your program does not handle or ignore this signal, your program will quit immediately. You can handle this in one of two ways:

  - Ignore the signal globally with the following line of code:

    ```
    signal(SIGPIPE, SIG_IGN);
    ```

  - Tell the socket not to send the signal in the first place with the following lines of code (substituting the variable containing your socket in place of `sock`):

    ```
    int value = 1;
    setsockopt(sock, SOL_SOCKET, SO_NOSIGPIPE, &value, sizeof(value));
    ```

    For maximum compatibility, you should set this flag on each incoming socket immediately after calling `accept` in addition to setting the flag on the listening socket itself.

- Use nonblocking sockets where possible.
- Keep the socket's send buffer full to the extent possible.
- Handle incoming data early and often to keep the socket's receive buffer empty.
- Support both IPv4 and IPv6.
- Check return values from socket reads and writes.

  Be prepared to handle `EAGAIN` and `EWOULDBLOCK` errors, which indicate that no data is available when reading, or that no output buffer space is available when writing. These errors are normal, non-fatal errors; your program should not close the connection when it gets them.

For an example of POSIX code that follows these rules, see Writing a TCP-Based Server in *Networking Programming Topics*.

Avoid Resolving DNS Names Before Connecting to a Host

The preferred way to connect to a host is with an API that accepts a DNS name, such as `CFHost` or `CFNetService`.

Although your program can resolve a DNS name and then connect to the resulting IP address, you should generally avoid doing so. DNS lookups usually return multiple IP addresses, and (at the application layer) it is not always obvious which IP address is best for connecting to the remote host. For example:

- Most modern computers and other mobile devices are multihomed. This means that they exist on more than one physical network at the same time. Your computer might be on Wi-Fi and Ethernet; your cellular phone might be on Wi-Fi and 3G; and so on. However, not all hosts are necessarily available over every connection.

  For example, the Remote app on your iPhone lets you control your Apple TV, but only over the Wi-Fi connection. The two devices cannot communicate with one another over your phone's cellular connection because your Apple TV has no public IP address.

- If both your device and the server you are connecting to have multiple IP addresses on different networks, the best IP address for connecting to the server may depend on which network the connection will pass through.

  For example, if your home media server has one IP address on your wired LAN and a second IP address on a Wi-Fi network, the operating system can often detect the performance difference and favor the faster, LAN-based IP address. By contrast, if your program looks up the IP address, it has approximately an equal chance of connecting to either IP address.

- If the server has both IPv4 and IPv6 protocols, it may not be reachable using both protocols. By using a host-name-based API, the operating system can try both simultaneously and use the first one that connects successfully. If your program looks up the IP address, it might not connect successfully, depending on which address it chose.

- If the DNS server is an older server that does not handle IPv6 and you request an `AAAA` (IPv6 address) record, synchronous DNS queries will block until the request times out (30 seconds by default). If you use an API that takes a hostname, the operating system hides this problem from you by issuing IPv4 and IPv6 requests in parallel, and then canceling the outstanding IPv6 lookup as soon as an IPv4 connection is established successfully.

- If a server is using Bonjour to advertise a service over a link-local IP address (because DHCP is not working on the network for some reason), if your app looks up that service and resolves it to an IP address, the resulting address will work only as long as the device retains that IP address. If the DHCP server comes online, the IP address may change, at which time your program will no longer be able to connect to the old address.

  If you connect using the advertised name instead, your program will continue to be able to connect even after the server finally obtains an IP address (so long as the computer or device running that program receives the updated advertisement).
- If the server is on the other side of an on-demand VPN that becomes available only when the user tries to access a whitelisted host, connecting by IP does not activate that VPN, which means that the host will never become reachable.

If you cannot avoid resolving a DNS name yourself, first check to see whether the `CFHost` API fulfills your requirements; it provides a list of addresses for a given host instead of just a single address. If the `CFHost` API does not meet your needs, use the DNS Service Discovery API.

For more information, read *Networking Concepts*, Writing a TCP-Based Server in *Networking Programming Topics*, *CFHost Reference*, and *DNS Service Discovery C Reference*. For sample code, see *SRVResolver*.

Do Not Use NSSocketPort (OS X) or NSFileHandle for General Socket Communication

Despite its name, the `NSSocketPort` class (available in OS X only) is not intended for general network communication. The `NSSocketPort` class is part of Cocoa's distributed objects system, which is intended for controlled communication between Cocoa applications on a single machine or on a local network. For more information on the distributed objects system, see *Distributed Objects Programming Topics*.

Similarly, the `NSFileHandle` class is not designed for general networking. The `NSFileHandle` class circumvents the standard networking stack, which carries the following drawbacks:

- Network connections made with `NSFileHandle` can be significantly less efficient than those made with the standard networking APIs.
- Historically, using `NSFileHandle` for networking has resulted in either extremely poor performance or strange, hard-to-debug failures.
- There is no straightforward way to use TLS authentication and encryption on connections made with `NSFileHandle`.
- In iOS, `NSFileHandle` does not automatically activate the device's cellular modem or on-demand VPN.

Instead, use `NSStream` for remote connections and `CFSocket` for listening. For details, see Writing a TCP-Based Server in *Networking Programming Topics*.

Supporting IPv6 DNS64/NAT64 Networks

With IPv4 address pool exhaustion imminent, enterprise and cellular providers are increasingly deploying IPv6 DNS64 and NAT64 networks. A DNS64/NAT64 network is an IPv6-only network that continues to provide access to IPv4 content through translation. Depending on the nature of your app, the transition has different implications:

- If you're writing a client-side app using high-level networking APIs such as `NSURLSession` and the CFNetwork frameworks and you connect by name, you should not need to change anything for your app to work with IPv6 addresses. If you aren't connecting by name, you probably should be. See Avoid Resolving DNS Names Before Connecting to a Host (page 54) to learn how. For information on CFNetwork, see *CFNetwork Framework Reference*.
- If you're writing a server-side app or other low-level networking app, you need to make sure your socket code works correctly with both IPv4 and IPv6 addresses. Refer to RFC4038: Application Aspects of IPv6 Transition.

What's Driving IPv6 Adoption

Major network service providers, including major cellular carriers in the the United States, are actively promoting and deploying IPv6. This is due to a variety of factors.

> **Note:** World IPv6 Launch is an organization that tracks deployment activity at a global scale. To see recent trends, visit the World IPv6 Launch website.

IPv4 Address Depletion

For decades, the world has known that IPv4 addresses would eventually be depleted. Technologies such as Classless Inter-Domain Routing (CIDR) and network address translation (NAT) helped delay the inevitable. However, on January 31, 2011, the top-level pool of Internet Assigned Numbers Authority (IANA) IPv4 addresses was officially exhausted. The American Registry for Internet Numbers (ARIN) is projected to run out of IPv4 addresses in the summer of 2015—a countdown is available here.

IPv6 More Efficient than IPv4

Aside from solving for the IPv4 depletion problem, IPv6 is more efficient than IPv4. For example, IPv6:

- Avoids the need for network address translation (NAT)
- Provides faster routing through the network by using simplified headers
- Prevents network fragmentation
- Avoids broadcasting for neighbor address resolution

4G Deployment

The fourth generation of mobile telecommunication technology (4G) is based on packet switching only. Due to the limited supply of IPv4 addresses, IPv6 support is required in order for 4G deployment to be scalable.

Multimedia Service Compatibility

IP Multimedia Core Network Subsystem (IMS) allows services such as multimedia SMS messaging and Voice over LTE (VoLTE) to be delivered over IP. The IMS used by some service providers is compatible with IPv6 only.

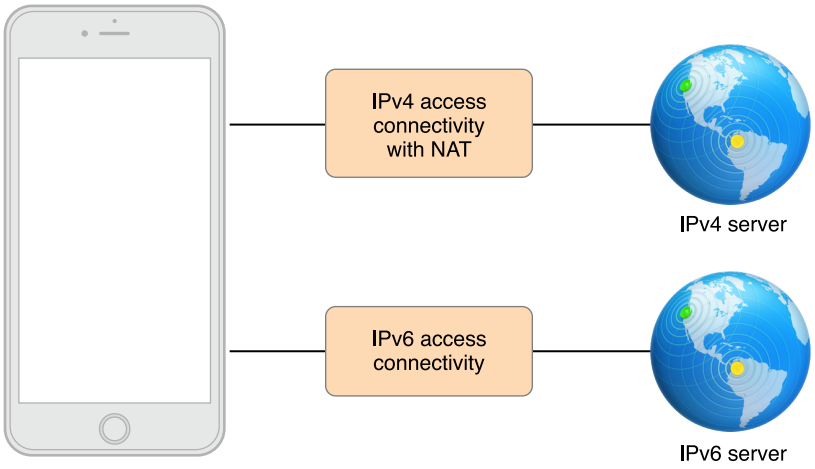
Cost

Service providers incur additional operational and administrative costs by continuing to support the legacy IPv4 network while the industry continues migrating to IPv6.

DNS64/NAT64 Transitional Workflow

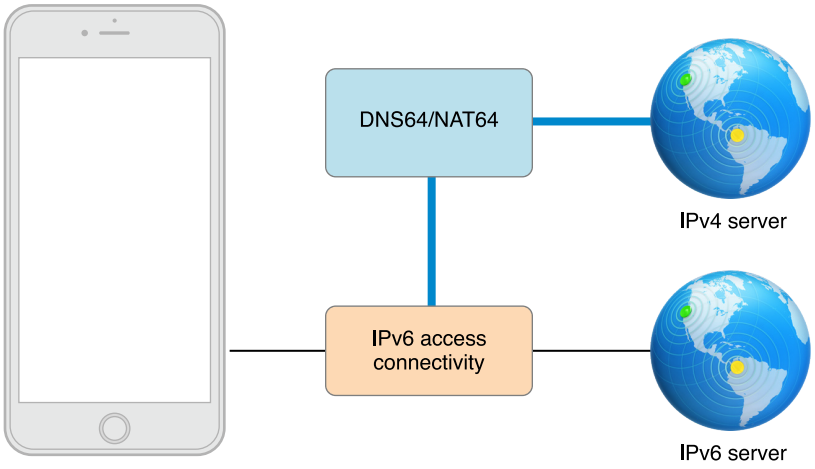
To help slow the depletion of IPv4 addresses, NAT was implemented in many IPv4 networks. Although this solution worked temporarily, it proved costly and fragile. Today, as more clients are using IPv6, providers must now support both IPv4 and IPv6. This is a costly endeavor.

Figure 10-1 A cellular network that provides separate IPv4 and IPv6 connectivity



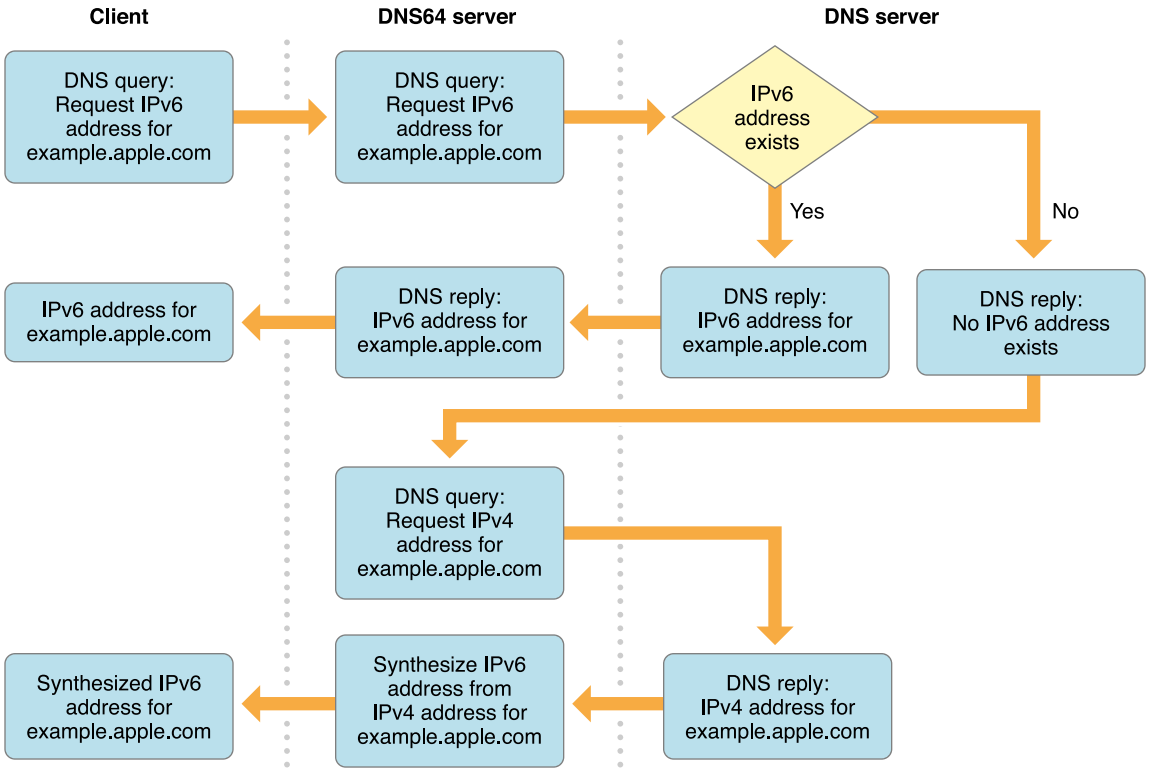
Ideally, providers want to drop support for the IPv4 network. However, doing so prevents clients from accessing IPv4 servers, which represent a significant portion of the Internet. To solve this problem, most major network providers are implementing a DNS64/NAT64 transitional workflow. This is an IPv6-only network that continues to provide access to IPv4 content through translation.

Figure 10-2 A cellular network that deploys an IPv6 network with DNS64 and NAT64



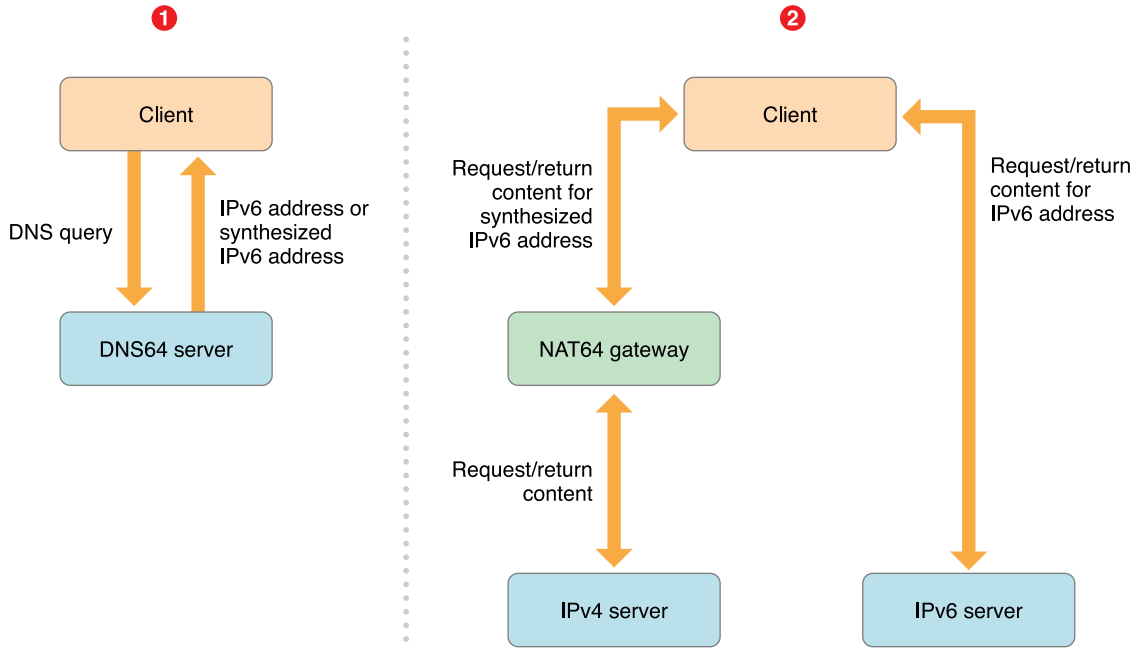
In this type of workflow, the client sends DNS queries to a DNS64 server, which requests IPv6 addresses from the DNS server. When an IPv6 address is found, it's passed back to the client immediately. However, when an IPv6 address isn't found, the DNS64 server requests an IPv4 address instead. The DNS64 server then synthesizes an IPv6 address by prefixing the IPv4 address, and passes that back to the client. In this regard, the client always receives an IPv6-ready address. See Figure 10-3.

Figure 10-3 DNS64 IPv4 to IPv6 translation process



When the client sends a request to a server, any IPv6 packets destined for synthesized addresses are automatically routed by the network through a NAT64 gateway. The gateway performs the IPv6-to-IPv4 address and protocol translation for the request. It also performs the IPv4 to IPv6 translation for the response from the server. See Figure 10-4.

Figure 10-4 Workflow of a DNS64/NAT64 transitional solution



IPv6 and App Store Requirements

Compatibility with IPv6 DNS64/NAT64 networks will be an App Store submission requirement, so it is essential that apps ensure compatibility. The good news is that the majority of apps are already IPv6-compatible. For these apps, it's still important to regularly test your app to watch for regressions. Apps that aren't IPv6-compatible may encounter problems when operating on DNS64/NAT64 networks. Fortunately, it's usually fairly simple to resolve these issues, as discussed throughout this chapter.

Common Barriers to Supporting IPv6

Several situations can prevent an app from supporting IPv6. The sections that follow describe how to resolve these problems.

- **IP address literals embedded in protocols.** Many communications protocols, such as Session Initiation Protocol (SIP), File Transfer Protocol (FTP), WebSockets, and Peer-to-Peer Protocol (P2PP), include IP address literals in protocol messages. For example, the `FTP` parameter commands `DATA PORT` and `PASSIVE`

exchange information that includes IP address literals. Similarly, IP address literals may appear in the values of SIP header fields, such as `To`, `From`, `Contact`, `Record-Route`, and `Via`. See Use High-Level Networking Frameworks (page 63) and Don't Use IP Address Literals (page 64).

- **IP address literals embedded in configuration files.** Configuration files often include IP address literals. See Don't Use IP Address Literals (page 64).
- **Network preflighting.** Many apps attempt to proactively check for an Internet connection or an active Wi-Fi connection by passing IP address literals to network reachability APIs. See Connect Without Preflight (page 64).
- **Using low-level networking APIs.** Some apps work directly with sockets and other raw network APIs such as `gethostbyname`, `gethostbyname2`, and `inet_aton`. These APIs are prone to misuse or they only support IPv4—for example, resolving hostnames for the `AF_INET` address family, rather than the `AF_UNSPEC` address family. See Use High-Level Networking Frameworks (page 63).
- **Using small address family storage containers.** Some apps and networking libraries use address storage containers—such as `uint32_t`, `in_addr`, and `sockaddr_in`—that are 32 bits or smaller. See Use Appropriately Sized Storage Containers (page 65).

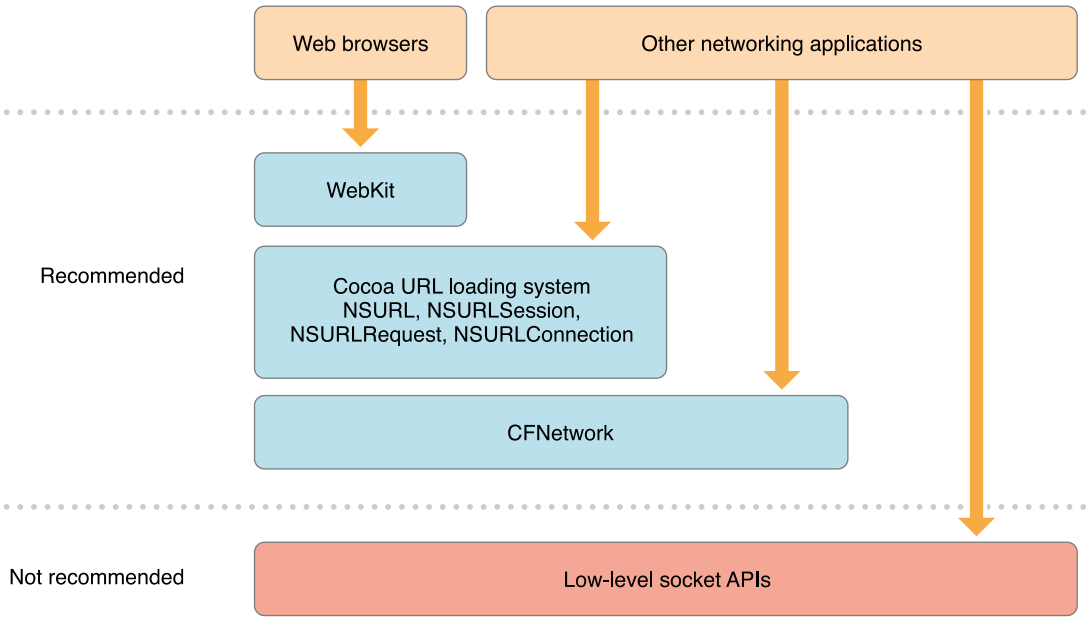
Ensuring IPv6 DNS64/NAT64 Compatibility

Adhere to the following guidelines to ensure IPv6 DNS64/NAT64 compatibility in your app.

Use High-Level Networking Frameworks

Apps requiring networking can be built upon high-level networking frameworks or low-level POSIX socket APIs. In most cases, the high-level frameworks are sufficient. They are capable, easy to use, and less prone to common pitfalls than the low-level APIs.

Figure 10-5 Networking frameworks and API layers



- **WebKit.** This framework provides a set of classes for displaying web content in windows, and implements browser features such as following links, managing a back-forward list, and managing a history of pages recently visited. WebKit simplifies the complicated process of loading webpages—that is, asynchronously requesting web content from an HTTP server where the response may arrive incrementally, in random order, or partially due to network errors. For more information, see *WebKit Framework Reference*.
- **Cocoa URL loading system.** This system is the easiest way to send and receive data over the network without providing an explicit IP address. Data is sent and received using one of several classes—such as `NSURLSession`, `NSURLRequest`, and `NSURLConnection`—that work with `NSURL` objects. `NSURL` objects let your app manipulate URLs and the resources they reference. Create an `NSURL` object by calling the `initWithString:` method and passing it a URL specifier. Call the `checkResourceIsReachableAndReturnError:` method of the `NSURL` class to check the reachability of a host. For more information, see *URL Session Programming Guide*.
- **CFNetwork.** This Core Services framework provides a library of abstractions for network protocols, which makes it easy to perform a variety of network tasks such as working with BSD sockets, resolving DNS hosts, and working with HTTP/HTTPS. To target a host without an explicit IP address, call the

`CFHostCreateWithName` method. To open a pair of TCP sockets to the host, call the `CFStreamCreatePairWithSocketToCFHost` method. For more information, see CFNetwork Concepts in *CFNetwork Programming Guide*.

If you do require the low-level socket APIs, follow the guidelines in RFC4038: Application Aspects of IPv6 Transition.

Note: *Getting Started with Networking, Internet, and Web* and *Networking Overview* provide detailed information on networking frameworks and APIs.

Don't Use IP Address Literals

Make sure you aren't passing IPv4 address literals in dot notation to APIs such as `getaddrinfo` and `SCNetworkReachabilityCreateWithName`. Instead, use high-level network frameworks and address-agnostic versions of APIs, such as `getaddrinfo` and `getnameinfo`, and pass them hostnames or fully qualified domain names (FQDNs). See `getaddrinfo(3) Mac OS X Developer Tools Manual Page` and `getnameinfo(3) Mac OS X Developer Tools Manual Page`.

Note: In iOS 9 and OS X 10.11 and later, `NSURLSession` and `CFNetwork` automatically synthesize IPv6 addresses from IPv4 literals locally on devices operating on DNS64/NAT64 networks. However, you should still work to rid your code of IP address literals.

Connect Without Preflight

The Reachability APIs (see *SCNetworkReachability Reference*) are intended for diagnostic purposes *after* identifying a connectivity issue. Many apps incorrectly use these APIs to proactively check for an Internet connection by calling the `SCNetworkReachabilityCreateWithAddress` method and passing it an IPv4 address of `0.0.0.0`, which indicates that there is a router on the network. However, the presence of a router doesn't guarantee that an Internet connection exists. In general, avoid preflighting network reachability. Just try to make a connection and gracefully handle failures. If you must check for network availability, avoid calling the `SCNetworkReachabilityCreateWithAddress` method. Call the `SCNetworkReachabilityCreateWithName` method and pass it a hostname instead.

Some apps also pass the `SCNetworkReachabilityCreateWithAddress` method an IPv4 address of `169.254.0.0`, a self-assigned link-local address, to check for an active Wi-Fi connection. To check for Wi-Fi or cellular connectivity, look for the network reachability flag `kSCNetworkReachabilityFlagsIsWWAN` instead.

Use Appropriately Sized Storage Containers

Use address storage containers, such as `sockaddr_storage`, that are large enough to store IPv6 addresses.

Check Source Code for IPv6 DNS64/NAT64 Incompatibilities

Check for and eliminate IPv4-specific APIs, such as:

- `inet_addr()`
- `inet_aton()`
- `inet_lnaof()`
- `inet_makeaddr()`
- `inet_netof()`
- `inet_network()`
- `inet_ntoa()`
- `inet_ntoa_r()`
- `bindresvport()`
- `getipv4sourcefilter()`
- `setipv4sourcefilter()`

If your code handles IPv4 types, make sure the IPv6 equivalents are handled too.

IPv4	IPv6
`AF_INET`	`AF_INET6`
`PF_INET`	`PF_INET6`
`struct in_addr`	`struct in_addr6`
`struct sockaddr_in`	`struct sockaddr_in6`
`kDNSServiceProtocol_IPv4`	`kDNSServiceProtocol_IPv6`

Use System APIs to Synthesize IPv6 Addresses

If your app needs to connect to an IPv4-only server without a DNS hostname, use `getaddrinfo` to resolve the IPv4 address literal. If the current network interface doesn't support IPv4, but supports IPv6, NAT64, and DNS64, performing this task will result in a synthesized IPv6 address.

Listing 10-1 shows how to resolve an IPv4 literal using `getaddrinfo`. Assuming you have an IPv4 address stored in memory as four bytes (such as `{192, 0, 2, 1}`), this example code converts it to a string (such as `"192.0.2.1"`), uses `getaddrinfo` to synthesize an IPv6 address (such as a `struct sockaddr_in6` containing the IPv6 address `"64:ff9b::192.0.2.1"`) and tries to connect to that IPv6 address.

Listing 10-1 Using `getaddrinfo` to resolve an IPv4 address literal

```
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <err.h>

    uint8_t ipv4[4] = {192, 0, 2, 1};
    struct addrinfo hints, *res, *res0;
    int error, s;
    const char *cause = NULL;

    char ipv4_str_buf[INET_ADDRSTRLEN] = { 0 };
    const char *ipv4_str = inet_ntop(AF_INET, &ipv4, ipv4_str_buf,
sizeof(ipv4_str_buf));

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = PF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_DEFAULT;
    error = getaddrinfo(ipv4_str, "http", &hints, &res0);
    if (error) {
        errx(1, "%s", gai_strerror(error));
        /*NOTREACHED*/
    }
    s = -1;
    for (res = res0; res; res = res->ai_next) {
        s = socket(res->ai_family, res->ai_socktype,
                   res->ai_protocol);
        if (s < 0) {
            cause = "socket";
```

```
            continue;
        }

        if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
            cause = "connect";
            close(s);
            s = -1;
            continue;
        }

        break;  /* okay we got one */
    }
    if (s < 0) {
        err(1, "%s", cause);
        /*NOTREACHED*/
    }
    freeaddrinfo(res0);
```

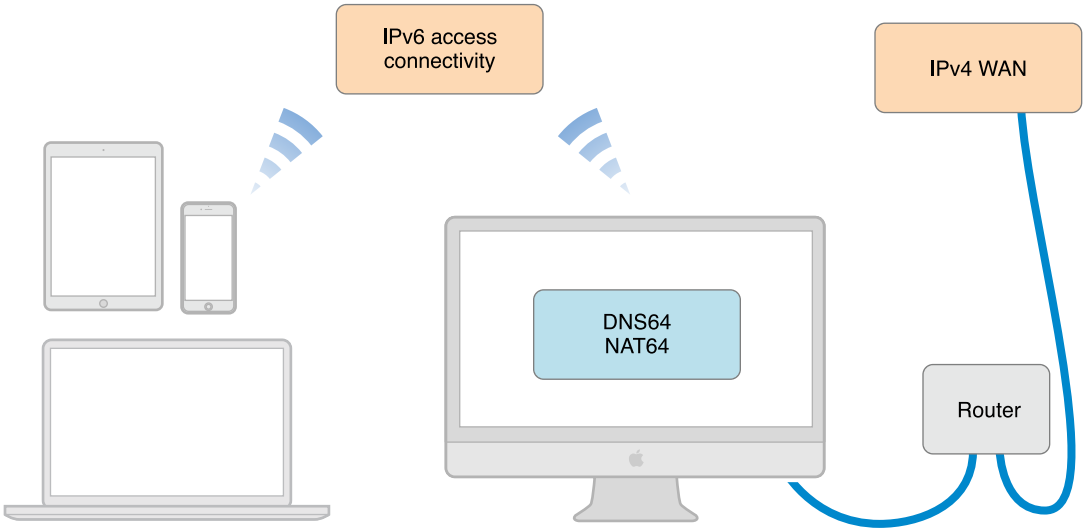
Note: The ability to synthesize IPv6 addresses was added to `getaddrinfo` in iOS 9.2 and OS X 10.11.2. However, leveraging it does not break compatibility with older system versions. See `getaddrinfo(3) Mac OS X Developer Tools Manual Page`.

Test for IPv6 DNS64/NAT64 Compatibility Regularly

The easiest way to test your app for IPv6 DNS64/NAT64 compatibility—which is the type of network most cellular carriers are deploying—is to set up a local IPv6 DNS64/NAT64 network with your Mac. You can then connect to this network from your other devices for testing purposes. See Figure 10-6.

Important: IPv6 DNS64/NAT64 network setup options are available in OS X 10.11 and higher. In addition, a Mac-Based IPv6 DNS64/NAT64 network is compatible with client devices that have implemented support for RFC6106: IPv6 Router Advertisement Options for DNS Configuration. If your test device is not an iOS or OS X device, make sure it supports this RFC. Note that, unlike DNS64/NAT64 workflows deployed by service providers, a Mac-Based IPv6 DNS64/NAT64 always generates synthesized IPv6 addresses. Therefore, it does not provide access to IPv6-only servers outside of your local network.

Figure 10-6 A local Mac-based IPv6 DNS64/NAT64 network



To set up a local IPv6 Wi-Fi network using your Mac

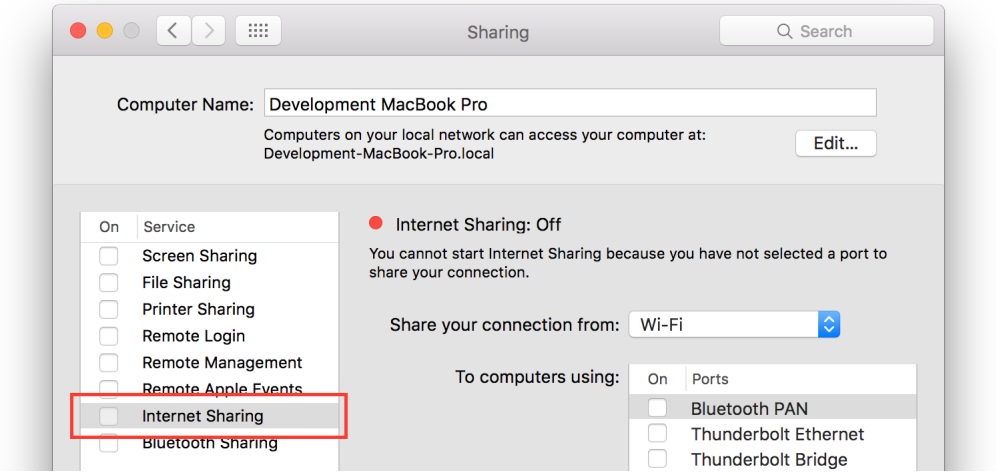
1. Make sure your Mac is connected to the Internet, *but not through Wi-Fi*.
2. Launch System Preferences from your Dock, LaunchPad, or the Apple menu.

3. Press the Option key and click Sharing. Don't release the Option key yet.

Figure 10-7 Opening Sharing preferences

4. Select Internet Sharing in the list of sharing services.

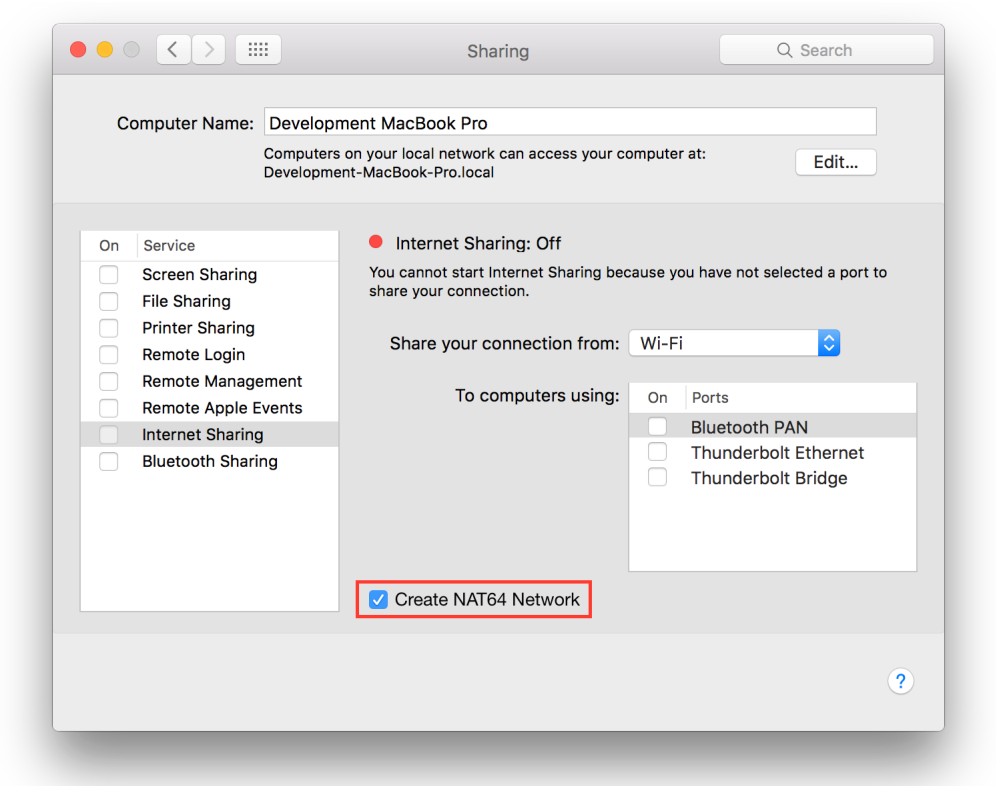
Figure 10-8 Configuring Internet sharing



5. Release the Option key.

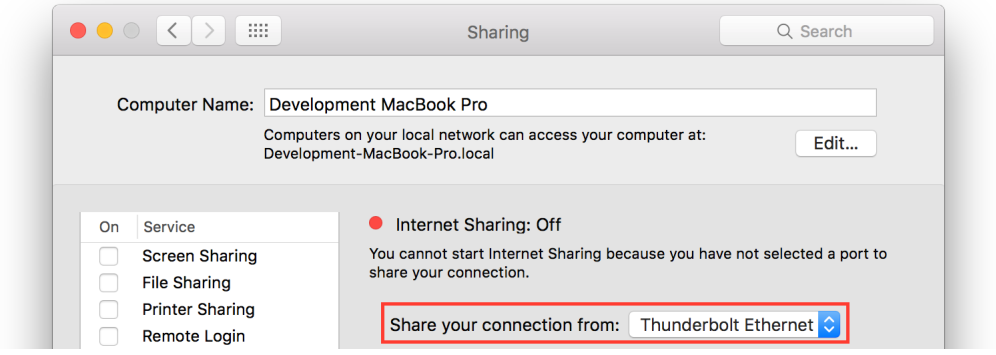
6. Select the Create NAT64 Network checkbox.

Figure 10-9 Enabling a local IPv6 NAT64 network



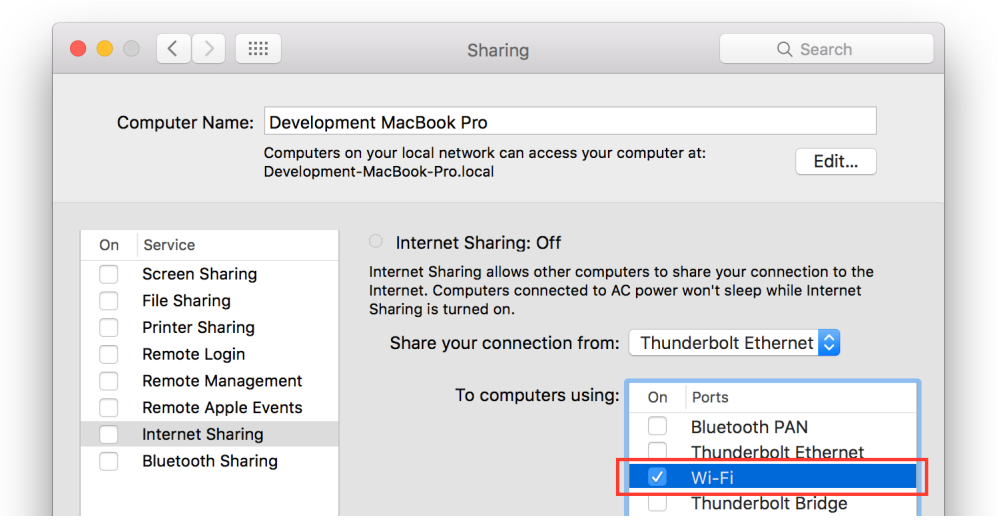
7. Choose the network interface that provides your Internet connection, such as Thunderbolt Ethernet.

Figure 10-10 Choosing a network interface to share



8. Select the Wi-Fi checkbox.

Figure 10-11 Enabling sharing over Wi-Fi



9. Click Wi-Fi Options, and configure the network name and security options for your network.

Figure 10-12 Accessing Wi-Fi network options

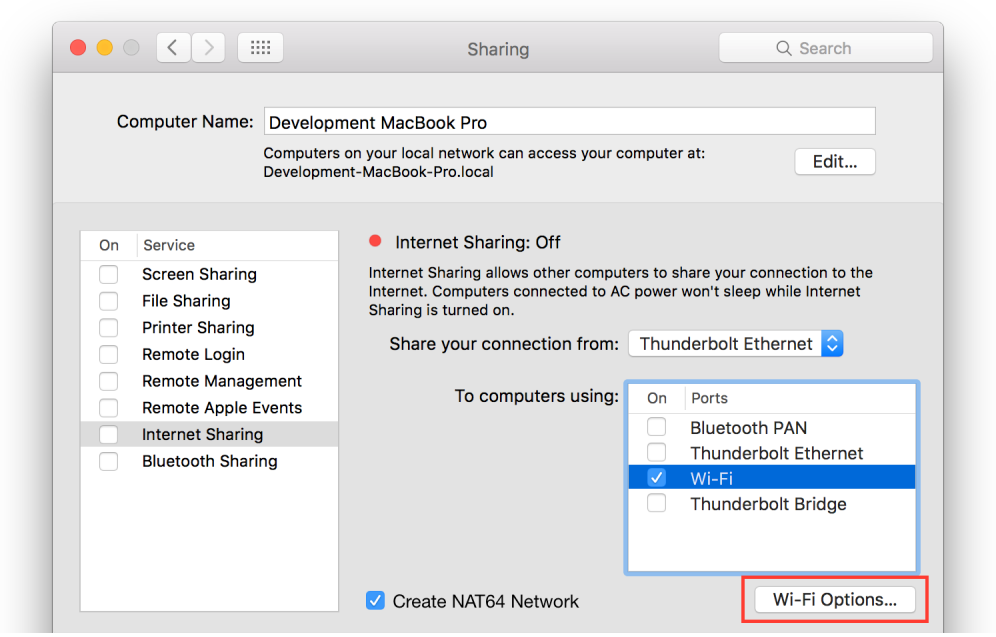
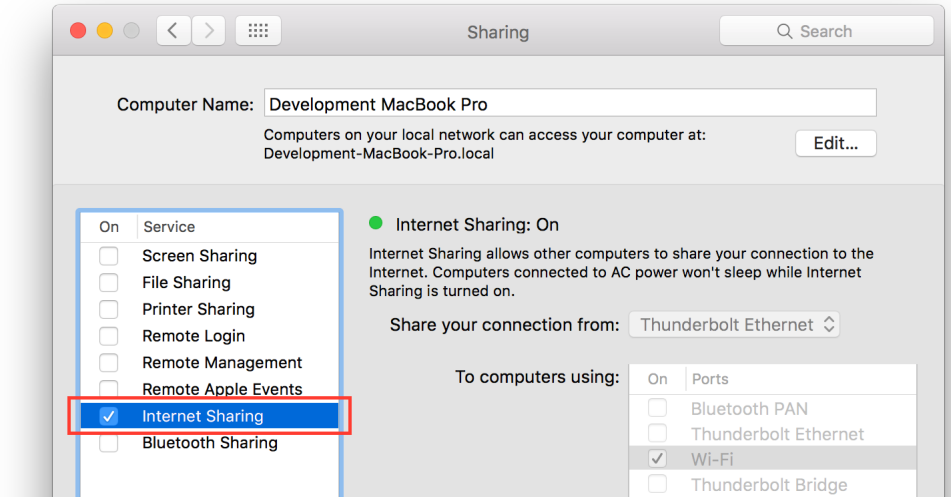


Figure 10-13 Setting up local Wi-Fi network options

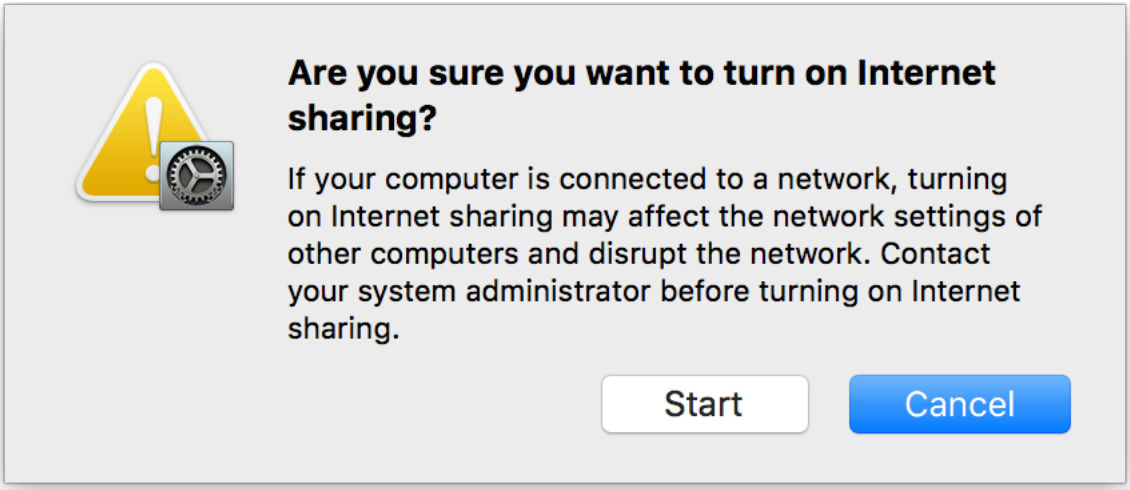
10. Select the Internet Sharing checkbox to enable your local network.

Figure 10-14 Enabling Internet sharing



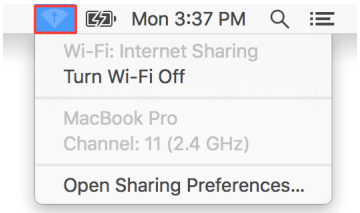
11. When prompted to confirm you want to begin sharing, click Start.

Figure 10-15 Starting Internet sharing



Once sharing is active, you should see a green status light and a label that says Internet Sharing: On. In the Wi-Fi menu, you will also see a small, faint arrow pointing up, indicating that Internet Sharing is enabled. You now have an IPv6 NAT64 network and can connect to it from other devices in order to test your app.

Figure 10-16 Internet sharing indicator



Important: To ensure that testing takes place strictly on the local IPv6 network, make sure your test devices don't have other active network interfaces. For example, if you are testing with an iOS device, make sure cellular service is disabled so you are only testing over Wi-Fi.

Resources

For more information on implementing networking, see:

- *Networking Programming Topics*

- *CFNetwork Programming Guide*
- *NSURLSession Class Reference*
- *WebKit Framework Reference*

For more information on the IPv6 transition, see:

- WWDC15 Video: Your App and Next Generation Networks
- RFC4038: Application Aspects of IPv6 Transition
- World IPv6 Launch website
- American Registry for Internet Numbers (ARIN) IPv4 Depletion Countdown

For technical issues encountered while transitioning to IPv6, see:

- Apple Developer Forums
- Developer Technical Support

Document Revision History

This table describes the changes to *Networking Overview*.

Date	Notes
2015-10-29	Updated to include information about using system APIs to synthesize IPv6 addresses.
2015-10-21	Updated to include expanded information about the IPv6 transition.
2014-03-10	Moved some content to URL Loading System Programming Guide.
2013-09-18	Corrected minor technical errors.
2013-01-28	Added information about how to prevent the use of cellular data.
2012-07-19	New document that provides a starting point for learning about networking in OS X and iOS.

Glossary

Address Resolution Protocol (ARP) A protocol for determining the hardware address of a computer or other device based on its IP address.

application layer The topmost layer of the networking protocol stack. This layer consists of data formats and protocols specific to a given application. For example, the HTTP (hypertext transport protocol) standard is an application-layer protocol.

broadcast address A special address that sends a packet simultaneously to every device on a local area network.

core router A router that provides service for major Internet backbone routes. Core routers are powerful devices that must handle a large volume of traffic and usually must manage a large number of simultaneous routes. Core routers participate in route advertisements to discover or announce changes in the network topology.

default gateway The default router used for outgoing traffic if there is no explicit route for the destination IP in the system's routing table.

domain name A human-readable name that identifies an Internet or intranet site; for example, developer.apple.com is a domain name. By resolving a domain name, an application can obtain a corresponding IP address that is suitable for sending data to that site.

edge router A router that provides connectivity between a customer site and an upstream ISP. Edge routers generally route between only two or three different networks, and thus usually do not participate in route advertisements.

encapsulation The act of wrapping one packet inside another packet (usually of a different type). For example, on a local area network, your IP packets are encapsulated within Ethernet packets. The Ethernet packets provide information about their destination within the local area network. The IP packets inside them provide information about what to do with the packets once they reach the public Internet.

fragmentation The process of breaking up a packet into smaller pieces to accommodate network connections with a smaller maximum packet size (referred to as the maximum transmission unit, or MTU).

header In the context of packets, the first part of a packet (before the actual payload) that contains information about where the packet should be sent. In the context of HTTP, a series of values that provide information about the content of a request or reply, such as the hostname, caching policies, and so on.

hop Any one of a series of physical links that make up the route from one host to another.

host Any device that is connected to a network. It may be a client computer, a server, a mobile phone, or even a network-attached printer.

hostname (or host name) A DNS name that points to a specific host (or a group of hosts that mimic a single host).

infrastructure device Any device that provides support for a network's basic operation—for example, a router, a Wi-Fi access point, or an Ethernet switch.

Internet Control Message Protocol (ICMP) A low-level networking protocol that provides out-of-band control messages that are used by the operating system when making TCP connections. ICMP is used mainly to deliver connection failure notifications—"connection refused" and "host unreachable" messages, for example. However, it is also used by some network diagnostic tools, such as `ping` and `traceroute`.

IP (Internet Protocol) layer The networking layer that provides basic transport of packets across the Internet. It sits above the physical layer (hardware interconnects) and below the transport layer (TCP and UDP, for example).

IP address A number that uniquely identifies a single host on the Internet (short for Internet Protocol address). An IP address can be in one of two forms: an IPv4 address or an IPv6 address.

IPv4 address An IP address consisting of four 8-bit numbers (for a total of 32 bits). For example, the IP address for developer.apple.com is 17.254.2.129.

IPv6 address An IP address consisting of eight groups of 16-bit hexadecimal numbers (for a total of 128 bits). If several groups in a row are all zero, you can omit those groups and replace them with a double colon (but only once per IP address). For example, the IPv6 address for example.com is 2001:500:88:200::10.

latency The amount of time it takes for a packet to reach its destination, usually measured in milliseconds. Latency is usually expressed as round-trip latency, which refers to the amount of time for a packet to reach its destination and for the response packet to reach the original host. Latency is important for two reasons. First, it increases the amount of time it takes to establish a connection. Second, it dramatically reduces performance when using protocols that require the client to wait for a response before sending subsequent requests.

link A physical connection between two hosts on a network (or a virtual connection that emulates a physical connection) with no intermediate routers (except for link-layer switches).

link layer The lowest layer of the network protocol stack. This layer provides support for the physical transport of packets from one host to another across a local area network or other physical link.

listening socket (or listen socket) A socket configured to listen for incoming connections.

Maximum Transmission Unit (MTU) The largest packet size that can be delivered across a particular link. The MTU is limited by the actual communication hardware, and usually represents the maximum payload size supported by the largest physical packet that the hardware supports. However, in some cases (such as gigabit Ethernet), the default MTU may be further limited in software to maintain backwards compatibility with legacy hardware that does not support larger packets.

multicast A special type of packet that is simultaneously delivered to a multitude of hosts on the network, but not to every host (broadcast).

neighbor discovery protocol (NDP) A protocol used by IPv6 over Ethernet to learn about other devices on the physical network. Among other

things, neighbor discovery can be used to learn the hardware addresses of nearby devices, discover routers and name servers, and determine information about upstream links, such as their Maximum Transmission Unit (MTU).

netblock See subnet.

netmask A collection of bits indicating which portion of an IPv4 address is the network part and which portion is the host part. If the network part of the destination address is the same as the network part of the source address, the two hosts are considered to be within the same subnet.

network address A special reserved address within each IPv4 network in which the host part is all zeros. This address was used by older operating systems as the broadcast address, so for historical compatibility reasons, this number is reserved.

network address translation (NAT) A form of packet rewriting performed by a firewall in which packets are modified to contain a different source or destination IP address before passing them on. NAT is most commonly used to make traffic from multiple devices appear to come from a single device, often for security or load balancing purposes.

network interface A piece of hardware (or virtual hardware) that represents the endpoint of a link.

packets A discrete unit of data that is sent across a computer network.

path MTU discovery A process by which one host determines the largest packet that can be sent to a destination without fragmenting it. This allows the host to fragment the data ahead of time, which prevents packets from potentially being fragmented more than once before reaching their final destination. Path MTU discovery works by sending packets with the "Don't Fragment" bit set. If any router along the path responds by sending an ICMP packet with the Fragmentation Needed bit set, the host then tries progressively smaller sizes until the packet reaches its destination successfully. See also Maximum Transmission Unit (MTU).

payload The data contents of a packet (as distinct from the structure of the packet itself).

physical layer See link layer.

port numbers A number that uniquely identifies a particular service on a given host. Port numbers are further divided according to whether they are TCP or UDP ports.

recursion The use of recursive queries. A recursive query asks the domain name server to perform recursion on the client's behalf. If the domain name server allows recursive queries, it then sends a query to the root name server asking which server knows the answer, then asks that server, and so on, until it reaches a server that actually knows the answer to the query. See also recursion.

route The path that packets take from one host to another host. If the two hosts are on the same physical network, the route consists of a single link; if not, it passes through one or more routers.

router A device that routes packets between two or more networks. A router determines which network should receive each packet based on a set of routing rules. Most routers also communicate with other routers to optimize those rules as network links are added and removed.

router address The IP address of your router.

routing The process of taking a packet on one physical network and retransmitting it on a different physical network, using a set of rules to determine which network should receive each packet. A device that performs routing is called a router.

shared network A network in which every packet is received by every device on the network. This is the opposite of a switched network.

subnet A range of IP addresses in which packets from one host can be sent directly to another host without going through an intermediate router.

switched network A physical network in which an infrastructure device (called a switch) directs packets based on their destination. This improves network performance by ensuring that only the hosts that need to receive a given packet actually see it. This is the opposite of a shared network.

trailer The last part of a packet (after the payload) that usually contains a checksum of the payload data.

Transmission Control Protocol (TCP) A transport-layer protocol that provides bidirectional, stream-based delivery of data, with flow control and delivery guarantees (automatic retry). Contrast with User Datagram Protocol (UDP).

transport layer The networking layer that sits on top of the IP layer and can provide such features as port numbers, delivery guarantees, flow control, and checksums. The two most common transport-layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

User Datagram Protocol (UDP) A transport-layer protocol that provides unidirectional, packet-based delivery of data, with best-effort delivery (no retransmission). Contrast with Transmission Control Protocol (TCP).

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010