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 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.

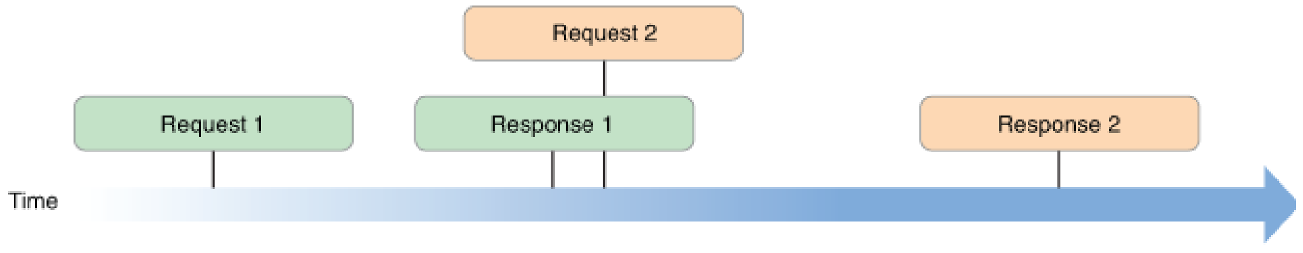
Design for High Latency

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.

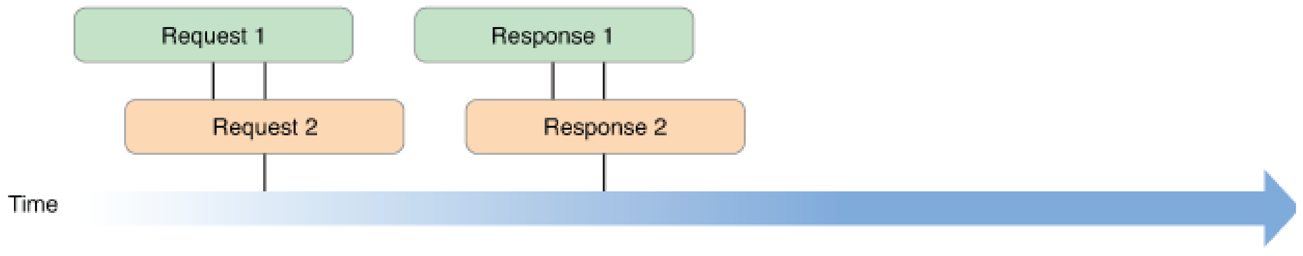
When an app makes multiple requests to one or more remote hosts, if it waits for the first request to return a result before making the second one, the connection latency becomes additive; the second request is penalized by the latency of the first request in addition to its own, the third request is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



Sending requests one at a time



Sending multiple requests at the same time

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.