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 and Using Sockets and Socket Streams.

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*.