High-Performance Networking for Mobile Applications

Best practices for efficiency, responsiveness, and reliability

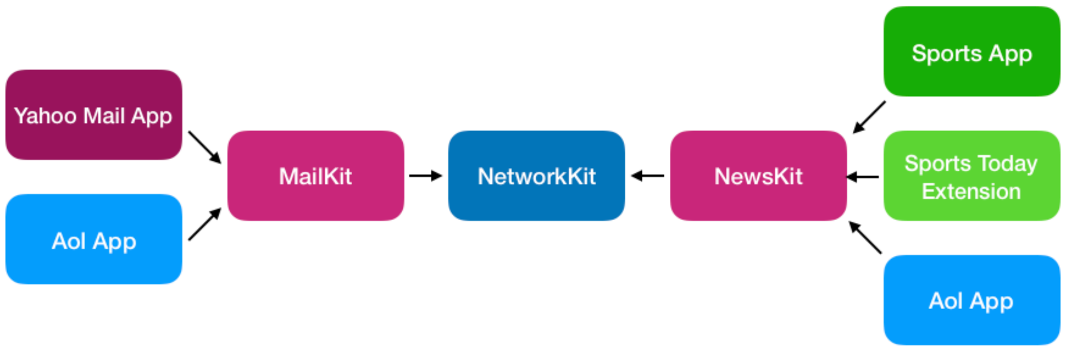
by Jason Howlin, Senior iOS Engineer, Mobile Shared Tech. July 2019.

Verizon Media iOS and Android apps communicate with backend APIs for data, such as news articles, video, and email. The networking layer of a mobile application is a core component and critical for two reasons: 1) the integrity and correctness of the app and user's data, and 2) the performance and efficiency of the app, including conserving the user's data plan and battery life. Managing this network communication is important to a great user experience and the app reliability.

Even though connectivity has improved over the years due to the increase in wi-fi hotspots and LTE connectivity, network communication should still be considered a precious resource. Users may have a finite amount of data to use, and spinning up a device's radios for networking is one of the most expensive operations you can perform. The goal should always be to use as little data as possible.

iOS provides frameworks to handle networking essentials, but there are a number of techniques and best practices to follow when building the networking layer of a mobile app. This paper presents techniques used in three of Verizon's iOS apps (Aol app, Yahoo Play app, and the DoublePlay news SDK), and how these practices improve performance, efficiency, and reliability.

Independent Frameworks



It is helpful to treat the networking in the app as an independent subsystem that is as isolated as possible, with the fewest number of dependencies. Think of the networking layer as an SDK, where your app is a consumer, but where it might be used in a number of different apps as well.

The trend in iOS is to extend app functionality using app extensions, such as a Today widget, and all app extensions are actually independent applications. Having networking and data models separated into their own framework is essential for code sharing.

DoublePlay SDK, recognizing the need for news inside of Today extensions or Notification service extensions, has separated the networking and data into a lightweight library, independent of the larger SDK and UI components.

Central Networking Manager

This is all it takes to send a network request on iOS:

```
let url = URL(string: “https://www.yahoo.com")
```

```
URLSession.shared.dataTask(with: url) { data, response, error in
    if let data = data { ... }
}.resume()
```

Because of the simplicity, it's easy to construct and send these requests from many different places in the application. But this quickly becomes unmanageable and inefficient. It's helpful to have one point for which all of the networking in your app passes through. Without coordination, one part of the app may not know what is happening in another part, resulting in duplicate work or duplicate error handling.

For example, in the Aol Mail app if a user needs to re-authenticate, then the next request to perform an action will return an error, requiring the user to re-login. This will cause an error to be displayed in the UI, and begin a re-login flow.

If there is no central manager to handle this (such as a NetworkManager type class), any number of background, simultaneous, or queued requests will all generate the same error, potentially prompting a number of error messages to be displayed, stacking on top of one another, resulting in a poor, confusing user experience.

A NetworkManager can avoid this by emitting one error and holding on to other failed requests for retry once the user has fixed their password issue.

A similar situation was faced in the Yahoo Play app, where a "crumb" identifier needed to be fetched before any other requests were issued. It helped to have a central location to make the crumb request, and queue other incoming requests until it was completed.

A network manager is also a great help when debugging, especially for developers new to the codebase. For any network request, a developer should be able to find one location where a breakpoint can be set to allow inspection of the request: the destination URL, any parameters attached, any authentication related information. A similar breakpoint can be set to examine the response from the server. Because this is such a common use case, it helps to enable logging at these points (which we'll discuss in more detail later).

It is not recommended to use a singleton for this class, rather a single instance that is passed using dependency injection. This provides greater control over when and where it is accessed, as well as avoiding shared state and side effects between unit test case runs.

Constructing Requests

When an app needs to make a network call for data, DoublePlay and Yahoo Play build a request object. This encapsulates the required query parameters, headers, and any body data, and ultimately is used to generate the object the system requires for a network request.

Writing specific functions with non-optional parameters that generate these requests help ensure all information required for the request is supplied, and can be unit tested for correctly generating the final URL.

Because of Swift language support for generics and functions as first-class types, you can even encode both then expected type returned from the server for a request, and a parser function to decode the JSON response into that type. Responses can be modeled using Swift's Result type, returning:

```
Result<(T:Codable, HTTPURLResponse), NetworkError>
```

Where Result is an enum with associated values: It has a success case, in which case it also provides a tuple of two values: a type T, which is generic for the type of parsed response expected (and constrained to be Codable), and an HTTP response code. The failure case's associated value is a custom network error.

Avoiding Duplicate Concurrent Requests

For any network request 1 in progress, if another part of the application makes the same request, it would be inefficient to issue the second request, mainly because we transfer twice the data, and request 2 will take a more time than necessary to complete.

A better approach is for the central networking manager to identify request 1 and request 2 as exactly the same, and since request 1 is in-flight, let request 2 wait for the results of request 1 and use them as its own, reporting the results to the issuer of request 2.

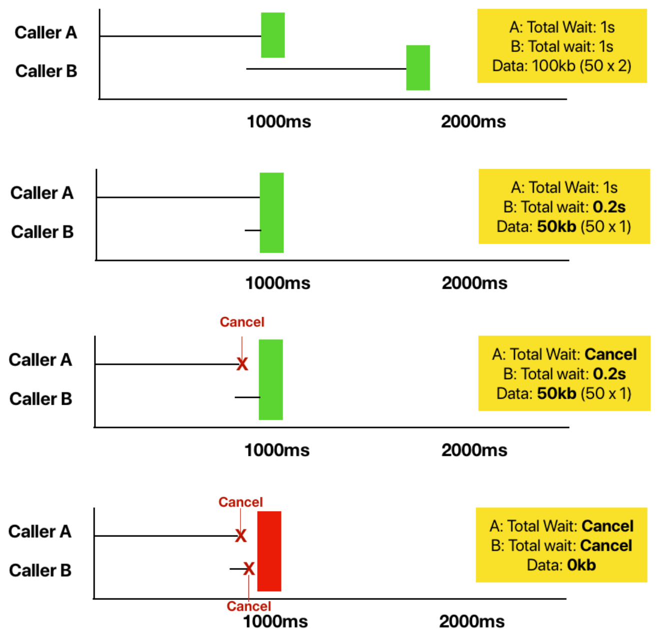
A typical pattern when networking on iOS is that requests are made along with a callback function. When the request completes, the function is invoked with the results of the request. In Swift, functions can easily be passed as parameters to other functions, and stored as values inside of data structures. Therefore it's easy to associate a number of callback functions with a request and store them in an in-memory data structure:

```
var completions = [RequestType:
[RequestUUID:CompletionFunction]]()
```

Where for any unique type of request, a number of callback functions can be stored, each with its own unique identifier. When the request completes, each callback is executed and provided the same Result.

**Due to the asynchronous nature of networking, care should be taken to synchronize adding and removing data to this structure by reading and writing exclusively using a serial DispatchQueue.*

If one networking manager is handling all requests, it can identify duplicates, ensure a second request isn't issued, and instead store the callback to be called when the inflight request completes.



1) Without deduping, requests overlap, we transfer twice the data, and B waits longer than needed. 2) We detect A & B are the same, so B waits for A to complete and uses the same results. 3) A initiated the request, B waits, A is then cancelled, yet we still need the request to complete to service B. 4) Here both requests were cancelled, so we can safely cancel the network request.

An important detail is that requests must be uniquely identifiable, and not only the URL or endpoint considered, but also any query parameters and post body data.

For example, if a request to the "sendMessage" endpoint arrives while we're in the process of sending a message, if we're dealing with two different messages it would be disastrous to not make the second request because we already have a "sendMessage" request inflight.

Cancelling Requests

To avoid unnecessary data transfer and to improve responsiveness, the application should be able to cancel any or all of network requests either inflight or queued up for execution. This may be because a user has signed out of an account, is performing a search and modified their search criteria, rendering the previous requests invalid, or has requested a large image to be downloaded, and not only do we want to avoid transferring the data, we also want to avoid any heavy image post-processing.

The iOS system frameworks make cancelling easy as well:

```
task.cancel()
```

However, a reference to the task itself is needed to call this code.

A central networking manager can, if a request is identifiable, keep a reference to the task and can cancel any inflight requests, using a structure:

```
var inflightTasks = [RequestType:URLSessionTask]()
```

Note that because we are deduplicating requests based on RequestType, we will never have more than one task for a single request.

Requests need to be reference counted. If there is one inflight request, and one waiter, if one cancels it should not impact the other. Only when a cancellation would result in no waiters (a reference count of 0) should the network request truly be cancelled.

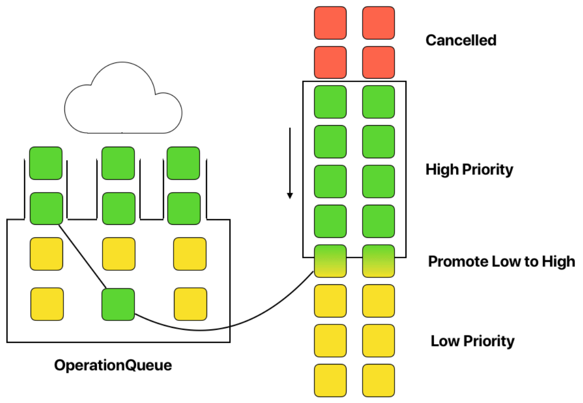
Prioritizing and Queueing Requests

Some requests are more important than others: background prefetching the latest 100 messages in a user's Inbox is a low-priority convenience, while executing a request to send a message is a high-priority requirement.

Sometimes a request is made with a low priority, and later becomes more important:

Consider scrolling through a list of 100 news articles: we lazily wait to download the images until an article is about to scroll on screen, however we might want to look ahead 10 or so articles and get a head start on downloading them, anticipating the user will soon be scrolling to them. The offscreen prefetching should be made with a low priority - we don't want to execute them ahead of the rows coming on screen.

However, if a user begins to scroll rapidly through the list, requests that began as a low-priority now suddenly become high-priority and should be executed immediately. And requests for articles that have scrolled offscreen are no longer a high priority, and should be cancelled.



As the user scrolls, we change the priority of image downloads about to appear onscreen, and cancel incomplete downloads moving offscreen.

The iOS 'Operation' and 'OperationQueue' classes provide a way to handle this. Operations wrap a unit of work - in our case a network request. The operation can then be assigned a priority, and submitted to the queue. The queue will pull them out in FIFO order by priority. Operations that have not executed and are in the ready state can have their priority changed or cancelled altogether.

The OperationQueue also allows us to limit the number of operations that execute concurrently. iOS will generally issue no more than 4-6 concurrent network requests, so we can set a similar number on our OperationQueue. This gives our networking layer far greater control.

Sometimes we need to make a number of requests, in order, and each one must finish before the next can begin. Take for example the process of the sign in process in the Aol app:

UserInfo -> Mailbox Info -> Settings -> All Folders -> Inbox Messages

Operations allow for creating dependencies. Each of these requests are wrapped in an Operation and allow us to ensure one doesn't start until the previous completes, and cancelling remaining tasks if any fail.

Authentication

If requests need to be authenticated, either by supplying a token in the header or supplying cookies in the request, one approach is for the networking layer to delegate out the signing of each request to the app.

The AOL Mail SDK was shared by two different applications, with different authentication methods. Delegating this work out to the consuming app avoided creating a hard dependency on any authentication libraries.

Another reason is that authentication libraries are often incompatible with other platforms. The Apple watch, for example, cannot present a web view for an OAuth sign in. Yet the Aol mail SDK has no limitation from running on the watch. However, the authentication is delegated out to the app, who stores and manages the login and tokens.

Another reason is that we always want the latest version of the authentication credentials - storing copies within the networking layer can lead to stale information.

It is common in the Yahoo ecosystem for apps to include cookies as a means of authentication. Very often requests go out and pick up the cookies stored on the system, placed there by the YAccountsSDK or the Phoenix SDK. This works, but has security risks, and is opaque to the developer.

The recommended approach, and the approach used in Yahoo Play and DoublePlay, is to disable using cookie storage for your API requests, and instead explicitly ask the accounts SDK for the latest cookies, refresh them if they are invalid, and insert them into the network request. It's best to do this right before the request goes out, so data does not become stale while queued up to execute.

Testing

One key to network testing is to break the network components into smaller pieces that can be tested.

Testing request construction: For a given request the app will typically supply a number of parameters. Writing a factory class that takes the parameters and vends a request object can be easily tested and assert the request url and any headers or body data are constructed correctly.

Parsing responses: Use JSON responses from the backend saved on disk in your test bundle. For any request, test by loading the data and decoding it into your model objects, and verifying properties are correct.

These type of tests do not make network calls - they assume the backend does its job correctly. They ensure that given a task to be performed in the app, our inputs to the backend are constructed correctly, and we handle the response.

Sometimes it is helpful to do a full integration test and make live requests in the test. There are pros and cons to this. The backend may be down or network connectivity bad leading to test failures when the client code is actually correct. It may be hard to find a set of known, static data to work with on the backend. However, it is a good approach when doing test driven development. Exploring new API from an integration test provides a way to execute the API before the UI is writen. Also, these tests can help identify any unexpected changes in real data, or even bugs on the backend.

Logging / Debugging

The Aol Mail SDK uses a number of features to aid in debugging, as well as offers logging to provide great insight into the current activity of the SDK. Logging takes careful thought and effort. Logging the wrong data or being too verbose creates noise and detracts from important information, such as critical errors. An SDK should be respectful of the consuming app by making logging opt-in, as well as not including a dependency on any particular logging framework.

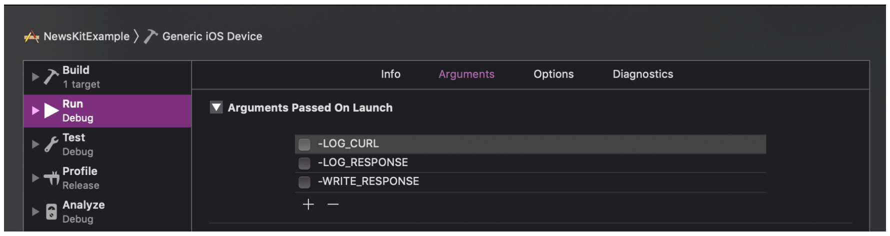
The Aol mail SDK uses a delegation pattern for logging: the consuming app sets a logging delegate where all messages are sent, and they can choose whether or not to log the information through their framework of choice. Aol app and Yahoo Play use Cocoa Lumberjack, which can write the logs to disk for debug and internal builds. These can be sent to developers from within the app if a user is experiencing a problem.

Types of log statements include when a network request is sent out, the endpoint, when the request completes, the duration of the request, and whether it succeeded or failed. This data formatted as a single line to keep it succinct and readable. This provides information about network health, and also gives the viewer a sense of what the user is doing and how the app responding to their actions.

```
START REQUEST: D35BE morenews-stream
END REQUEST: D35BE morenews-stream took 210 ms. SUCCESS
```

Additional debugging features include the ability to log a curl statement for a request, as well as log the complete JSON response to the console. Another option writes the JSON response to disk, which allows the SDK developers to capture responses, and add them to the app to use as mock data for unit tests.

These features are turned on and off inside the SDK by using launch arguments, so it is easy to enable without changing any code.



Arguments Passed On Launch

```
-LOG_CURL
-LOG_RESPONSE
-WRITE_RESPONSE
```

Supporting Long-Lived and Background Tasks

The iOS networking classes provide a way to ensure long-running or critical network tasks complete even if the app crashes, is put in the background, or is force-quit by the user. Handling these types of requests are different than typical request handling due to the lack of context - a particular request might finish the next day. We need to be able to associate the completion of a network response with our original intent some point far after the request was made. The app may have been restarted, so any context around the request that was stored in memory is now gone.

The Aol mail SDK uses this for sending e-mails with attachments, which can be large and take time to complete. If the user backgrounds the app, it is critical the action completes. To keep track of inflight message sends, information is saved to the database. When the app is notified at some future point that the network operation completed, we can look the task up in the database and mark it completed.

Looking Ahead

Apple recently added the Combine framework to iOS, which adds a rich set of functional reactive programming constructs, such as Observers, Publishers, and Futures, that bring a more declarative and composable way of performing asynchronous networking. There even exist specific publishers to send signals about networking events.

Futures help to model asynchronous code in a synchronous manner, by working with objects that exist now, but are populated with a value at some point in the future. This can improve code readability, and since futures can be shared by multiple objects waiting for the same results, reduce request duplication.

To underscore the need to be respectful of user’s data, Apple itself is building a “Low Data Mode” into iOS 13. When apps such as Aol Mail detect this mode, they should adapt accordingly, such as turning off mailbox prefetching, choosing to download low-res rather than high-res mail attachment images, and reducing the frequency of inbox syncing.

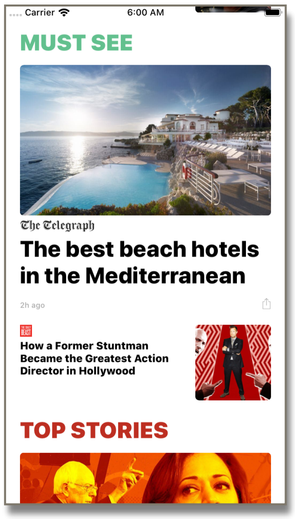
Sample Code

To demonstrate the techniques described in this paper, a sample iOS project has been created on GitHub. There is a reusable *NetworkKit* framework, and built on top of that is a sample *NewsKit* framework. The sample app uses the *NewsKit* framework (using NCP for data) to display a list of articles.

The sample app also includes a sample *ImageFetcherController* framework, which the UI uses to demonstrate the prefetching of images with priority described in the paper.

Note: the project requires Xcode 11, as the dependencies use the new Swift Package Manager integration.

https://git.ouroath.com/jhowlin/NewsKit


Carrier
6:00 AM
MUST SEE
The Telegraph
The best beach hotels in the Mediterranean
2h ago
How a Former Stuntman Became the Greatest Action Director in Hollywood
TOP STORIES

Sample code on GitHub demonstrating techniques discussed in this paper.