Working with HTTP Live Streaming

HTTP Live Streaming (HLS) is the ideal way to deliver media to your playback apps. With HLS, you can serve multiple media streams at different bit rates, and your playback client dynamically selects the appropriate streams as network bandwidth changes. This ensures that you're always delivering the best quality content given the user's current network conditions. This chapter looks at how to leverage the unique features and capabilities of HLS in your playback apps.

Playing Offline HLS Content

Starting with iOS 10, you can use AVFoundation to download HTTP Live Streaming assets to an iOS device. This new capability allows users to download and store HLS movies on their devices while they have access to a fast, reliable network, and watch them later without a network connection. With the introduction of this capability, HLS becomes even more versatile by minimizing the impact of inconsistent network availability on the user experience.

AVFoundation in iOS introduces several new classes to support downloading HLS assets for offline use. The following sections discuss these classes and the basic workflow used to add this capability to your app.

Preparing to Download

You use an instance of `AVAssetDownloadURLSession` to manage the execution of asset downloads. This is a subclass of `NSURLSession` that shares much of its functionality, but is used specifically for creating and executing asset download tasks. Like you do with an `NSURLSession`, you create an `AVAssetDownloadURLSession` by passing it an `NSURLSessionConfiguration` that defines its base configuration settings. This session configuration must be a *background* configuration so that asset downloads can continue while your app is in the background. When you create an `AVAssetDownloadURLSession` instance, you also pass it a reference to an object adopting the `AVAssetDownloadDelegate` protocol and an `NSOperationQueue` object. The download session will update its delegate with the download progress by invoking the delegate's methods on the specified queue.

```
func setupAssetDownload() {
    // Create new background session configuration.
    configuration = URLSessionConfiguration.background(withIdentifier: downloadIdentifier)

    // Create a new AVAssetDownloadURLSession with background configuration, delegate, and
queue
    downloadSession = AVAssetDownloadURLSession(configuration: configuration,
                                                assetDownloadDelegate: self,
                                                delegateQueue: OperationQueue.main)
}
```

After you create and configure the download session, you use it to create instances of `AVAssetDownloadTask` using the session's `assetDownloadTaskWithURLAsset:assetTitle:assetArtworkData:options:` method. You provide this method the `AVURLAsset` you want to download along with a title, optional artwork, and a dictionary of download options. You can use the options dictionary to target a particular variant bit rate or a specific media selection to download. If no options are specified, the highest quality variants of the user's primary audio and video content are downloaded.

```
func setupAssetDownload() {
    ...
    // Previous AVAssetDownloadURLSession configuration
    ...

    let url = // HLS Asset URL
```

```
    let asset = AVURLAsset(url: url)

    // Create new AVAssetDownloadTask for the desired asset
    let downloadTask = downloadSession.makeAssetDownloadTask(asset: asset,
                                                             assetTitle: assetTitle,
                                                             assetArtworkData: nil,
                                                             options: nil)
    // Start task and begin download
    downloadTask?.resume()
}
```

`AVAssetDownloadTask` inherits from `NSURLSessionTask`, which means you can suspend or cancel the download task using its `suspend` and `cancel` methods, respectively. In the case where a download is canceled, and there is no intention of resuming it, your app is responsible for deleting the portion of the asset already downloaded to a user's device.

Because download tasks can continue executing in a background process, you should account for cases where tasks are still in progress if your app was terminated while in the background. Upon application startup, you can use the standard features of the `NSURLSession` APIs to restore the state of any pending tasks. To do so, you create a new `NSURLSessionConfiguration` instance using the session configuration identifier with which you *originally* started these tasks, and recreate your `AVAssetDownloadURLSession`. You use the session's `getTasksWithCompletionHandler:` method to find any pending tasks and restore the state of your user interface, as shown below:

```
func restorePendingDownloads() {
    // Create session configuration with ORIGINAL download identifier
    configuration = URLSessionConfiguration.background(withIdentifier: downloadIdentifier)

    // Create a new AVAssetDownloadURLSession
    downloadSession = AVAssetDownloadURLSession(configuration: configuration,
                                                assetDownloadDelegate: self,
                                                delegateQueue: OperationQueue.main)

    // Grab all the pending tasks associated with the downloadSession
    downloadSession.getAllTasks { tasksArray in
        // For each task, restore the state in the app
        for task in tasksArray {
            guard let downloadTask = task as? AVAssetDownloadTask else { break }
            // Restore asset, progress indicators, state, etc...
            let asset = downloadTask.urlAsset
        }
    }
}
```

Monitoring the Download Progress

While the asset is downloading, you can monitor its progress by implementing the download delegate's `URLSession:assetDownloadTask:didLoadTimeRange:totalTimeRangesLoaded:timeRangeExpectedToLoad:` method. Unlike with other `NSURLSession` APIs, asset download progress is expressed in terms of loaded time ranges rather than bytes. You can calculate the asset's download progress using the time range values returned in this callback, as shown in the following example:

```
func urlSession(_ session: URLSession, assetDownloadTask: AVAssetDownloadTask, didLoad
timeRange: CMTimeRange, totalTimeRangesLoaded loadedTimeRanges: [NSValue],
timeRangeExpectedToLoad: CMTimeRange) {
    var percentComplete = 0.0
    // Iterate through the loaded time ranges
```

```
    for value in loadedTimeRanges {
        // Unwrap the CMTimeRange from the NSValue
        let loadedTimeRange = value.timeRangeValue
        // Calculate the percentage of the total expected asset duration
        percentComplete += loadedTimeRange.duration.seconds /
timeRangeExpectedToLoad.duration.seconds
    }
    percentComplete *= 100
    // Update UI state: post notification, update KVO state, invoke callback, etc.
}
```

Saving the Download Location

When the asset download is finished, either because the asset was successfully downloaded to the user's device or because the download task was canceled, the delegate's `URLSession:assetDownloadTask:didFinishDownloadingToURL:` method is called, providing the local file URL of the downloaded asset. Save a persistent reference to the asset's *relative path* so you can locate it at a later time:

```
func urlSession(_ session: URLSession, assetDownloadTask: AVAssetDownloadTask,
didFinishDownloadingTo location: URL) {
    // Do not move the asset from the download location
    UserDefaults.standard.set(location.relativePath, forKey: "assetPath")
}
```

You'll use this reference to recreate the asset for playback at a later time or delete it when the user would like to remove it from the device. Unlike with the `URLSession:downloadTask:didFinishDownloadingToURL:` method of `NSURLSessionDownloadDelegate`, clients should *not* move downloaded assets. The management of the downloaded asset is largely under the system's control, and the URL passed to this method represents the final location of the asset bundle on disk.

Important: Downloaded HLS assets are stored on disk in a private bundle format. This bundle format may change over time, and developers should not attempt to access or store files within the bundle directly, but should instead use AVFoundation and other iOS APIs to interact with downloaded assets.

Note: HLS asset downloads are automatically excluded from iCloud backups with the `NSURLIsExcludedFromBackupKey` key.

Downloading Additional Media Selections

You can update downloaded assets with additional audio and video variants or alternative media selections. This capability is useful if the originally downloaded movie does not contain the highest quality video bit rate available on the server or if a user would like to add supplementary audio or subtitle selections to the downloaded asset.

`AVAssetDownloadTask` downloads a single media-selection set. During the initial asset download, the user's default media selections—their primary audio and video tracks—are downloaded. If additional media selections such as subtitles, closed captions, or alternative audio tracks are found, the session delegate's `URLSession:assetDownloadTask:didResolveMediaSelection:` method is called, indicating that additional media selections exist on the server. To download additional media selections, save a reference to this resolved `AVMediaSelection` object so you can create subsequent download tasks to be executed serially.

```
func urlSession(_ session: URLSession, assetDownloadTask: AVAssetDownloadTask, didResolve
resolvedMediaSelection: AVMediaSelection) {
    // Store away for later retrieval when main asset download is complete
    // mediaSelectionMap is defined as: [AVAssetDownloadTask : AVMediaSelection]()
    mediaSelectionMap[assetDownloadTask] = resolvedMediaSelection
```

```
}
```

Before you download additional media selections, determine what has already been cached to disk. An offline asset provides an associated `AVAssetCache` object that you use to access the state of the asset's cached media. Using the `AVAsset` methods discussed in Selecting Media Options, you determine which media selections are available for this asset and use the asset cache to determine which values are available offline. The following method provides a way for you to find all audible and legible options that haven't yet been cached locally:

```
func nextMediaSelection(_ asset: AVURLAsset) -> (mediaSelectionGroup:
AVMediaSelectionGroup?,
                                                 mediaSelectionOption:
AVMediaSelectionOption?) {


    // If the specified asset has not associated asset cache, return nil tuple
    guard let assetCache = asset.assetCache else {
        return (nil, nil)
    }


    // Iterate through audible and legible characteristics to find associated groups for
asset
    for characteristic in [AVMediaCharacteristicAudible, AVMediaCharacteristicLegible] {


        if let mediaSelectionGroup = asset.mediaSelectionGroup(forMediaCharacteristic:
characteristic) {


            // Determine which offline media selection options exist for this asset
            let savedOptions = assetCache.mediaSelectionOptions(in: mediaSelectionGroup)


            // If there are still media options to download...
            if savedOptions.count < mediaSelectionGroup.options.count {
                for option in mediaSelectionGroup.options {
                    if !savedOptions.contains(option) {
                        // This option hasn't been downloaded. Return it so it can be.
                        return (mediaSelectionGroup, option)
                    }
                }
            }
        }
    }
    // At this point all media options have been downloaded.
    return (nil, nil)
}
```

This method retrieves the asset's available `AVMediaSelectionGroup` objects associated with the audible and legible characteristics and determines which of their `AVMediaSelectionOption` objects have already been downloaded. If it finds a new media-selection option that has not been downloaded, it returns that group-option pair in a tuple to the caller. You can use this method to help with the process of downloading additional media selections in the delegate's `URLSession:task:didCompleteWithError:` method, as shown in the following example:

Note: The following code example shows how to download all available audible and legible options. In most cases, you'd download only the media options specifically requested by the user.

```
func urlSession(_ session: URLSession, task: URLSessionTask, didCompleteWithError error:
Error?) {
```

```
    guard error == nil else { return }
    guard let task = task as? AVAssetDownloadTask else { return }

    // Determine the next available AVMediaSelectionOption to download
    let mediaSelectionPair = nextMediaSelection(task.urlAsset)

    // If an undownloaded media selection option exists in the group...
    if let group = mediaSelectionPair.mediaSelectionGroup,
           option = mediaSelectionPair.mediaSelectionOption {

        // Exit early if no corresponding AVMediaSelection exists for the current task
        guard let originalMediaSelection = mediaSelectionMap[task] else { return }

        // Create a mutable copy and select the media selection option in the media
selection group
        let mediaSelection = originalMediaSelection.mutableCopy() as!
AVMutableMediaSelection
        mediaSelection.select(option, in: group)

        // Create a new download task with this media selection in its options
        let options = [AVAssetDownloadTaskMediaSelectionKey: mediaSelection]
        let task = downloadSession.makeAssetDownloadTask(asset: task.urlAsset,
                                                         assetTitle: assetTitle,
                                                         assetArtworkData: nil,
                                                         options: options)

        // Start media selection download
        task?.resume()

    } else {
        // All media selection downloads complete
    }
}
```

Playing Offline Assets

After a download has been initiated, an app can simultaneously start playing an asset by creating an `AVPlayerItem` instance with the same asset instance used to initialize the `AVAssetDownloadTask` as shown in the following example:

```
func downloadAndPlayAsset(_ asset: AVURLAsset) {
    // Create new AVAssetDownloadTask for the desired asset
    // Passing a nil options value indicates the highest available bitrate should be
downloaded
    let downloadTask = downloadSession.makeAssetDownloadTask(asset: asset,
                                                             assetTitle: assetTitle,
                                                             assetArtworkData: nil,
                                                             options: nil)!
    // Start task
    downloadTask.resume()

    // Create standard playback items and begin playback
    let playerItem = AVPlayerItem(asset: downloadTask.urlAsset)
```

```
    player = AVPlayer(playerItem: playerItem)
    player.play()
}
```

When a user is concurrently downloading and playing an asset, it's possible that some portion of the video will be played at a lower quality than was specified in the download task's configuration. This can happen if network bandwidth constraints prevent streaming at the quality requested for download. When this situation occurs, `AVAssetDownloadURLSession` continues beyond the asset playback time, until all of the media segments at the requested quality are downloaded. After `AVAssetDownloadURLSession` is finished, the asset on disk will contain video at the requested quality level for the entire movie.

Whenever possible, reuse the same asset instance for playback as was used to configure the download task. This approach works well in the scenario described above, but what do you when the download is complete and the original asset reference or its download task no longer exists? In this case, you need to initialize a new asset for playback by creating a URL for the relative path you saved in Saving the Download Location. This URL provides the local reference to the asset as stored on the file system, as shown in the following example:

```
func playOfflineAsset() {
    guard let assetPath = UserDefaults.standard.value(forKey: "assetPath") as? String else
{
        // Present Error: No offline version of this asset available
        return
    }
    let baseURL = URL(fileURLWithPath: NSHomeDirectory())
    let assetURL = baseURL.appendingPathComponent(assetPath)
    let asset = AVURLAsset(url: assetURL)
    if let cache = asset.assetCache, cache.isPlayableOffline {
        // Set up player item and player and begin playback
    } else {
        // Present Error: No playable version of this asset exists offline
    }
}
```

This example retrieves the stored relative path and creates a file URL to initialize a new `AVURLAsset` instance. It tests to ensure that the asset has an associated asset cache and that at least one rendition of the asset is playable offline. Your app should check for the availability of downloaded assets and handle missing assets gracefully.

Note: When an iOS device is not connected to a network, the only options available for playback are those stored on the user's device. Apps must prevent a user from selecting unavailable options by limiting the choices available through their user interface to options present in the downloaded asset.

If the iOS device has a network connection, and an asset is configured to play back a variant or rendition that is not available in the downloaded asset, the playback engine streams the requested variant or rendition from the server, and plays it back in combination with the pieces of the asset that are stored locally.

Important: In extreme low-disk-space conditions, the operating system may automatically delete downloaded assets. Before you present to the user that an asset is available for playback, verify that the asset exists and is playable offline.

Managing the Asset Life Cycle

When you add offline HLS functionality to your app, you're responsible for managing the life cycle of assets downloaded to a user's iOS device. Ensure that your app provides a way for users to see the list of assets that are permanently stored on their device, including the size of each asset, and a way for users to delete assets when they need to free up disk space. Your app should also provide the appropriate UI for users to distinguish between assets stored locally on a device and assets available in the cloud.

You delete downloaded HLS assets using the `removeItemAtURL:error:` method of `NSFileManager`, passing it the asset's local URL, as shown below:

```
func deleteOfflineAsset() {
    do {
        let userDefaults = UserDefaults.standard
        if let assetPath = userDefaults.value(forKey: "assetPath") as? String {
            let baseURL = URL(fileURLWithPath: NSHomeDirectory())
            let assetURL = baseURL.appendingPathComponent(assetPath)
            try FileManager.default.removeItem(at: assetURL)
            userDefaults.removeObject(forKey: "assetPath")
        }
    } catch {
        print("An error occured deleting offline asset: \(error)")
    }
}
```

Observing Network Access and Error Logging

`AVPlayerItem` has a number of informational properties, such `loadedTimeRanges` and `playbackLikelyToKeepUp`, that can help you determine its current playback state. It also gives you access to the details of its lower-level state through two logging facilities found in its `accessLog` and `errorLog` properties. These logs provide additional information that can be used for offline analysis when working with HLS assets.

The access log is a running log of all network-related access that occurs while playing an asset from a remote host. `AVPlayerItemAccessLog` collects these events as they occur, providing you with insight into the player item's activity. The complete collection of log events is retrieved using the log's `events` property. This returns an array of `AVPlayerItemAccessLogEvent` objects, each representing a unique log entry. You can retrieve a number of useful details from this entry, such as the URI of the currently playing variant stream, the number of stalls encountered, and the duration watched. To be notified as new entries are written to the access log, you register to observe notifications of type `AVPlayerItemNewAccessLogEntryNotification`.

Similar to the access log, `AVPlayerItem` also provides an error log to access error information encountered during playback. `AVPlayerItemErrorLog` maintains the accumulated collection of error events modeled by the `AVPlayerItemErrorLogEvent` class. Each event represents a unique log entry and provides details such as the playback session ID, error status codes, and error status comments. As with `AVPlayerItemAccessLog`, you can be notified as new entries are written to the error log by registering to observe notifications of type `AVPlayerItemNewErrorLogEntryNotification`.

Because the access and error logs are intended primarily for offline analysis, it's easy to create a complete snapshot of the log in a textual format conforming to the W3C Extended Log File Format (see http://www.w3.org/pub/WWW/TR/WD-logfile.html). Both logs provide `extendedLogData` and `extendedLogDataStringEncoding` properties, making it easy to create a string version of the logs content:

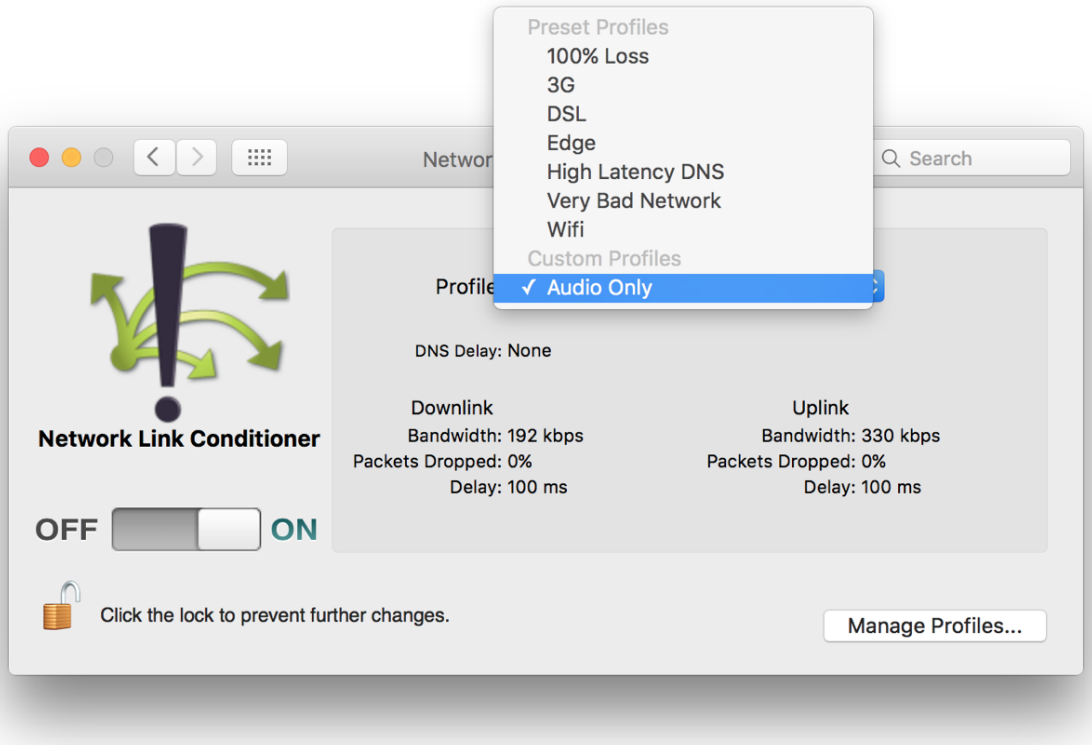
```
if let log = playerItem.accessLog() {
    let data = log.extendedLogData()!
    let encoding = String.Encoding(rawValue: log.extendedLogDataStringEncoding)
    let offlineLog = String(data: data, encoding: encoding)
    // process log
}
```

Testing with Network Link Conditioner

AVFoundation makes it easy to play HLS content in your app. The framework's playback classes handle most of the hard work for you, but you should test how your app responds as network conditions change. A utility called Network Link Conditioner can help with this testing.

Network Link Conditioner is available for iOS, tvOS, and macOS and makes it easy for you to simulate varying network conditions (see Figure 6-1).

Figure 6-1 Network Link Conditioner in macOS



This utility lets you easily switch between different network performance presets to ensure that your playback behavior is working as expected. In iOS and tvOS, you can find this utility under the Developer menu in Settings. In macOS, you can download this utility by choosing Xcode > Open Developer Tool > More Developer Tools. This selection takes you to the Downloads area of developer.apple.com; the utility is available as part of the "Additional Tools for Xcode" package.