AuthenTec
Downloadable Agent for iOS
With native player

User Manual

Table of Contents

Index of Figures

1. Foreword

This document is the user documentation for Downloadable Agent for iOS with native media player, hereafter to be referred to as “the Agent”.

This document provides a guide to integrating with the Agent components and its API. The target audience is developers integrating against the Agent API.

Readers of this manual are expected to have some prior knowledge of development for the iOS platform, the Xcode IDE and a thorough understanding of the Objective-C language in general.

2. Terminology and Conventions

For the purposes of the present document, the following terms and definitions apply.

2.1 Definitions

ARM	ARM is the 32-bits instruction set architecture supported by the processors of all iOS devices up to date, with ARMv6 and ARMv7 variants.
Device	Any device manufactured by Apple that runs the iOS operating system and is included in the list of devices supported by the Agent.
iOS	Originally known as "iPhone OS", refers to the operating system that powers all Apple mobile and TV devices.
Simulator	The iOS simulator included with the iOS SDK.
Player	Apple's QuickTime based native media player that is included in iOS.
HTTP Live Streaming	An HTTP-based adaptive media streaming communications protocol created by Apple.
Smooth Streaming	An IIS Media Services extension that enables adaptive streaming of media to clients over HTTP.
PIFF	The Protected Interoperable File Format defines a standard multimedia file format for delivery and playback of multimedia content.
ISMV	IIS Smooth Streaming Video file, a fragmented MPEG-4 file that follows the PIFF specifications.
PlayReady	A DRM technology created by Microsoft for portable devices.
SOAP	Protocol specification for exchanging structured information in the form of XML payloads.
Static library	An archive that contains a set of objects compiled for one or several different architectures and is to be statically linked with a stand-alone executable.
Protected library	Refers to a static library that has been compiled with some protection technologies, such as anti-tampering or anti-debugging.

2.2 Abbreviations

ARM	Advanced RISC Machine
DRM	Digital Rights Management
GDB	GNU Debugger
MIME	Multipurpose Internet Mail Extensions
SOAP	Simple Object Access Protocol

2.3 Conventions

Filenames and directories are displayed in italics, e.g. */usr/local/bin*

Pieces of example code in Objective-C are enclosed in text boxes with syntax highlighting. In some cases the example may exclude certain intermediate lines of code which are assumed to be already known by the user, in order to improve the readability. This is represented by three dots enclosed in square brackets.

```
// create an URL
NSURL *url = [NSURL URLWithString:@"http://www.authentec.com"];

// verify the URL
[...]
```

Informational information is indicated as shown below, with an (i) icon and accompanying text. The information indicated by the icon is informational.

Some informational text

Information which is critical to the running of the system and which, if not paid attention to, may cause failures in the system is indicated by the (!) icon and accompanying text. The information indicated by the icon is a warning and should be considered before acting on.

Some text which carries extra importance

3. Overview

AuthenTec Downloadable Agent is a flexible and easy-to-use client side software library that can be integrated with your application in order to provide the ability to play multimedia content protected with DRM. This allows you to impose certain restrictions to how/when the content can be played or who can play it.

The Agent is pre-integrated with the platform's native media player. This has many advantages over using a 3rd party media player, such as for example that it can make full use of the device's hardware acceleration to decode and render video, which results in smother playback and allows for a higher quality content and a better battery life.

The user experience is also important. By using the native player to play DRM protected content you achieve a consistent and pleasant experience for your users with a simple user interface they are already used to. You may also choose to hide all the visible DRM details away from the user, making it completely transparent.

The current version of the Agent supports adaptive streaming protocols such as **HTTP Live Streaming** (a.k.a. HLS) and **Microsoft Smooth Streaming**, and also static content, either pre-downloaded to the device or streamed to progressively.

3.1 Architecture of the client

The following diagram represents the internal architecture of an application that integrates with the Agent and its communication with the external entities.

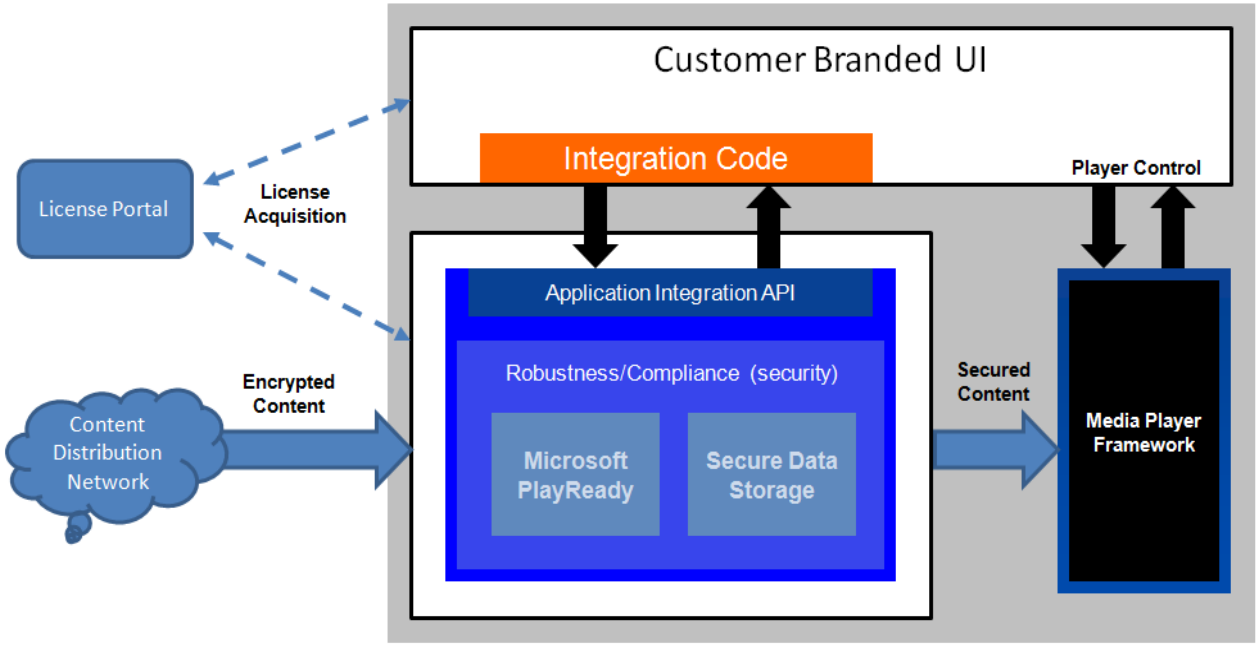

Figure 1 - Architecture of the client application

In general terms we can distinguish the following elements:

- **The application created and branded by the customer**
  This is the application the Agent is integrated with. It's fully created and customized by the customer; the Agent will never display any UI element on screen. The application may also interact with the native player using its public API, if it wishes to do so.

  AuthenTec provides the full source code of an example application which should serve as a reference for the programmer when integrating your own app with the Agent. See Section 4.

- **The integration API**
  The Agent's API exposes a set of methods that allow the application to easily acquire licenses for protected content and prepare the native player to play it. This API is provided by a static library that your application must be linked against.

- **The Media Player and/or the AVFoundation Framework**
  These frameworks are part of the iOS SDK and allow the application to render media content in different ways, with optional customizations applied to the player, such as for example the size and position of video rendering view or the playback controls. Only when used in conjunction with the Agent these frameworks are able to play content protected with PlayReady DRM.

- **The Content Distribution Network**
  The network of servers that host the encrypted content to be served either via streaming or downloaded to the device.

- **The PlayReady License Portal/Server**
  This is the server that issues licenses for the content on demand of the client. The communication between the server and the client typically takes place using the SOAP protocol.

This document will only describe the integration of your application with the Agent API. The Media Player framework and the external entities are out of the scope of this document and will only be referenced where appropriate.

3.2 A few words about security

Out of the box, the iOS platform offers a quite secure environment to deploy your applications. The users have very limited access and can hardly modify or alter the application in any way the developer did not originally intend to. There exists however several publicly available and easy to use techniques that allow the users to easily break into the security of the device and obtain root access to the underlying operating system. This process is popularly known as "jailbreaking" and is believed to be spread out over about a 10% of the devices.

With this in mind, AuthenTec has implemented several security measures into the Agent in order to protect it against attacks that try to remove the restrictions imposed by the DRM or to produce unlocked copies of the content. Despite the technical limitations imposed by the operating system on what application's level code can and cannot do, we are quite confident to have achieved a good level of protection.

This is a brief summary of the security measures we've put in place in our solution:

- The content is delivered encrypted end-to-end, from your content distribution network all the way up to the native media player.
- The sensitive data (keys, licenses, etc) is stored in an encrypted database on disk.

- Communications with the license server are encrypted and digitally signed.
- Several anti-tampering protections have been implemented in our SDK:
  - Integrity checks
  - Anti-debugger checks
  - Code obfuscation

Note: The anti-tampering features are only available in the protected version of the Agent library.

For more in deep information please refer to the Security Details PDF included with the documentation.

3.3 Supported devices / iOS versions

The following devices and iOS versions are supported:

- Devices with an ARMv7 CPU: iPhone 3GS, iPhone 4, iPhone 4S, iPad, iPad2, iPod Touch 3rd and 4th generation.
- iOS versions: 4.0 and later (iOS 5.0 is the latest major version available as of the writing of this document).

Older iOS devices such as the first and second generation iPhones (iPhone 2G, iPhone 3G) are not supported due to their older CPU which only supports the ARMv6 architecture.

3.4 Supported content formats

The following is a brief list of the high level content formats that can be played using the Agent in combination with the iOS native media player:

- ✓ **HTTP Live Streaming**
  This adaptive streaming format is natively supported by the iOS media player, but the encryption method described in the official specifications draft does not provide a secure key delivery mechanism, neither the possibility to enforce playback restriction whatsoever. To overcome these weaknesses the Agent adds support for HLS content encrypted with PlayReady DRM.
- ✓ **Smooth Streaming**
  This is another form of adaptive streaming technology over HTTP created by Microsoft. The Smooth Streaming protocol is not natively supported by the iOS media player. With the Agent you can also overcome this limitation and play your regular Smooth Streaming content –both protected and unprotected- with the native player, as well as perform any necessary license management, in exactly the same way you would play any other natively supported format.
- ✓ **PIFF files**
  Smooth Streaming uses ISMV files as inputs, which are fragmented MPEG-4 video files that follow the PIFF specifications. With the Agent you can also play these .ismv files individually, without adaptive streaming involved. You may choose between two different playback scenarios:
  - Local files: An .ismv file that has been either bundled with the application or fully downloaded to the device's file system can be played straight away by the Agent, even in offline mode (provided the content is not DRM protected or a valid license is already installed).
  - Progressive download: An .ismv file hosted in any standard web server (not necessarily IIS)

can also be progressively downloaded and played by the Agent (and optionally cached as well for offline playback).

For all the types of content formats described above the only DRM schema supported is **Microsoft PlayReady DRM**. AuthenTec provides as a separate package a batch encryption tool that can quickly and efficiently protect all your previously encoded media assets. Please contact your AuthenTec representative if you wish to buy or obtain an evaluation package.

Please read on to learn about the supported video and audio codec's, limitations and encoding recommendations for the best playback experience.

3.4.1 HTTP Live Streaming

HLS is currently the only adaptive streaming format natively supported by the iOS media player, therefore AuthenTec suggests to always follow Apple's recommendations and best practices for creating and deploying HTTP Live Streaming media for iOS devices.

The following documents are a much recommended read:

- HTTP Live Streaming Overview
- Best Practices for Creating and Deploying HTTP Live Streaming Media for the iPhone and iPad

When PlayReady DRM is applied to the assets, the following criteria must also be met:

- Each transport stream segment (.ts file) must be encrypted with PlayReady Envelope and thus contain a full PlayReady header embedded.
- Only one encryption key must be used to protect an entire media asset, including each and every different stream available in the master playlist. Key switching within the same streamed content is currently not supported. Different assets can and should use different encryption keys.
- The playlist files produced by your media segmenter tool or encoder do not need to be modified. No extra tags or extensions are required by the Agent to consume DRM protected content.
- If you wish to include CEA-608 closed captions in your streams you must do so before encrypting the assets.

Protecting your assets with DRM adds very little overhead, both in terms of resources and processing power. The Agent is able to download, decrypt and securely deliver the content to the media player without any degradation in quality or performance, which is why the same encoding guidelines used for clear text content can be applied for protected content.

3.4.2 Smooth Streaming

Smooth Streaming streams are converted on the fly to HLS-compatible streams by the Agent in order for the iOS native player to be able to play them. Therefore similar encoding recommendations in terms of bit-rate, resolution, frame rate, etc apply in this case, with the following important remarks:

- Only the **H.264** video codec is supported, either Baseline Profile or Main profile, depending on your target devices (see Apple recommendations in the previous section). Other codecs such as WMV are not supported.

- Use **AAC Low Complexity** for audio.
- Streams with multiple audio tracks: the Agent supports pre-selecting a particular audio track. If none has been selected it will then default to the top most audio stream in the Smooth Streaming client manifest.
- It is recommended to increase the default fragment size to a value between 3 and 9 seconds in length. This can be accomplished via the "key frame interval" setting in Expression Encoder 4 Pro. By doing so the playback experience might improve, especially in low or variable bandwidth conditions.
- AuthenTec has successfully tested HD streams of up to 5 Mbps with a resolution of 720p, although the results may vary depending on the hardware of the device (for example, an iPad 2 will be able to play higher quality streams than a first generation iPad).

Certain features of the Smooth Streaming protocol are currently not supported:

- Subtitles: any subtitle stream will simply be ignored by the Agent. Support for this feature is planned in the coming versions.
- Audio-only or video-only streams: This is currently not supported and might cause a non-deterministic behavior in the Agent and/or the player.

PlayReady encrypted streams are fully supported by the Agent. DRM schemas other than PlayReady are currently not supported though.

3.4.3 PIFF Files

PIFF files (.ismv) can be played by the Agent either directly from the local file system (i.e. as resource files bundled with or provisioned somehow to your iOS application) or downloaded progressively via HTTP.

The same settings and recommendations defined for the Smooth Streaming use case in the previous section are applicable for PIFF files.

Some extra considerations to take into account:

- The .ismv files should contain both an audio and a video stream; otherwise only video or only audio will be reproduced. Having the audio encoded in a separate file (.isma) is currently not supported by the Agent.
- Should an .ismv file contain more than one video stream, the Agent will try to play that with the highest bit-rate only. No adaptive switching will be performed.
- For the progressive download scenario, the web server might need to be configured to allow direct access to the .ismv files. This is typically the case for Microsoft IIS instances with the default configuration. Byte-range request must also be supported.
- For the local file scenario, the media files may be pre-bundled with the application, downloaded from the device after the installation or provisioned via iTunes to the application's user space.
- No client or server manifests (such as the ones generated for Smooth Streaming) are required to play .ismv files.
- You must use the content caching API should you wish to allow offline playback of progressively downloaded content.

4. Reference Implementation

Included in the release package is AuthenTec's reference implementation of a simple iOS application that makes use of the Agent API to play some samples of protected content, for both adaptive streaming and static content. This application, named "SampleApp", is provided in the form of an Xcode project and includes heavily commented source code on those parts where the Agent API is used. The SampleApp sports a very simple user interface on purpose, in order to minimize the amount of cluttering in the code and allow the developer to focus on the important part: the integration with the Agent.

Note that this example application does not necessarily reflect the best or the only way to use the Agent API, but it is intended to serve as a reference only. Your implementation will most likely vary to accommodate your own needs.

4.1 Building the SampleApp project

The first step is to copy the Agent's static libraries and headers included with the Agent SDK package into the SampleApp project folder. To do this, copy all the static libraries (.a files), the finalization files (.fin) and the entire `include` folder into the `DownloadableAgentLib` folder within the SampleApp project. Open up Xcode and ensure that none of the dependencies are in red. Otherwise you have probably not copied the missing file(s) or did it to the wrong folder.

The SampleApp project contains 2 build targets:

- **SampleApp**: This target builds the application for any iOS device supported as well as for the simulator platform.
- **SampleApp-Protected:** This target builds the application with the protected library. That restricts the supported platforms to iOS devices only and should not be attempted to be run while connected to a debugger (it will crash intentionally). Please note that this target requires some manual configuration to build successfully. See Section 5.1.3 for more information.

4.2 Architecture of the application

The entry point is the application's delegate class (`SampleAppAppDelegate)`, which performs the PlayReady model certificate provisioning (see Section 5.2) and loads up the main view. The view controller for the main view is the `SampleAppViewController` class, which also contains all the integration code to interact with the Agent's API and the native player. This view controller is also father to the other controllers:

- The `ConsoleLogViewController` manages the view that displays the log messages broadcasted by the Agent's logging subsystem, for easier debugging/testing while the device is detached from the Xcode Console. This view can be triggered by tapping on the little "i" icon on the lower right corner of the application's home screen.
- The `CachedContentViewController` manages the view that shows the status of the cache for those files that have been or are to be stored in cache in order to allow for offline playback.
- The `AVPlayerViewController` manages the playback of content with an `AVPlayer` (AVFoundation framework) as opposed to using the default `MPMoviePlayerController` (MediaPlayer framework).

Finally, a small open source library for displaying modal waiting screens is also used, such as for example when a license acquisition is in progress. This functionality is implemented by the `MBProgressHUD` class.

4.3 Using the SampleApp

The usage of the SampleApp is quite simple. At the application's home screen, follow these steps to play a piece of sample content:

1. Either type the URL of a supported content (e.g. the URL for the root playlist of an encrypted HTTP Live Streaming feed or for a Smooth Streaming manifest) or select one of the content URLs from the list of content samples provided by AuthenTec (tap the list-icon next to the address box to make the picker view pop up).
2. Press the "Play" button to begin the playback procedure, which includes checking first whether or not the content is protected and a valid license is installed.
3. If it is deemed necessary to acquire a license in order to play the selected content, the SampleApp will prompt the user for consent before proceeding to do so.
4. Once and if the license acquisition is completed successfully, the playback will automatically start right afterwards. Only full screen playback is implemented in the SampleApp at the moment.

You may also use the switch on the top right corner of the screen to optionally use the `AVPlayer` to play the content. We've implemented a very simple UI on top of it that may be a good starting point for you to develop your own and probably more complex UI for the player, should you choose to go with the AVFoundation framework.

Certain contents can also be cached onto disk for offline playback later on, provided that a valid license is installed or that the content is not encrypted. To demo the Download & Play Offline feature follow these steps:

1. Either type a URL or select one from the content list for a piece of content that supports caching for offline playback, such as is the case for remotely hosted PIFF files.
2. Tap the "Download & Play Offline" button to reveal another view that displays how much of the content, if any, is already cached on disk.
3. Use the on-screen buttons to start/pause/resume the caching process or to purge the entire cache.
4. Switch back to the main screen to initiate playback for this content URL. The fragments of the content that are cached on disk will be automatically used by the player. The fragments that are not will be fetched and cached should a working data connection be available.

Finally you may tap the "Reset DRM Database" button to remove any installed licenses regardless of the content URL that is currently selected.

5. Integration Guide

The following sections describe how to integrate the Downloadable Agent API in your project in order to easily play media content protected with PlayReady DRM.

The instructions on this guide correspond to the version 4.2 of Xcode. For other versions the location and/or names of the elements may vary slightly.

5.1 Project Setup

Please follow these steps in order to begin the integration of the Agent into your Xcode project.

5.1.1 Adding the Agent libraries as dependencies

Start by creating a new folder inside your project's root folder, where you'll be copying the static library and the header files. Name it `DownloadableAgentLib` for instance (the actual name chosen is not relevant).

Extract and copy into that folder the following items from the Agent's SDK package:

- The file `libDownloadableAgent/libDownloadableAgent.a`
- The entire folder `libDownloadableAgent/include`

The file `libDownloadableAgent.a` is a universal (multi-architecture) static library which contains binaries compiled for the iOS device platform (armv7) and the iOS simulator platform (i386). The `include` folder contains all the public header files for the library.

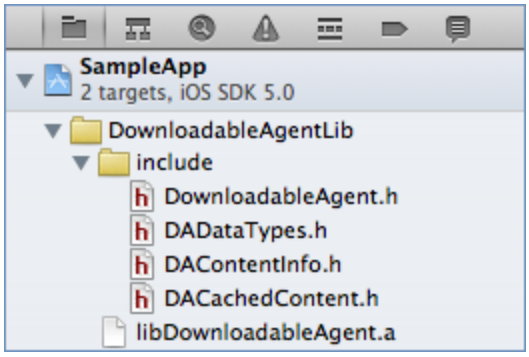

Figure 2 - Library and headers added to the project

Next, open up your project in Xcode; Click on "*File->Add Files to [your project name]*" and select the folder where you added all the files above. Make sure you have selected all the existing build targets where the Agent shall be used before clicking the "Add" button. The result should be something similar to what is shown in the **Error! Reference source not found.**.

The Agent's static library also depends on the following iOS frameworks, all of which you'll need to add to your project, should you not be using them already:

- Foundation.framework
- MediaPlayer.framework
- AVFoundation.framework
- CFNetwork.framework
- Security.framework
- SystemConfiguration.framework
- MobileCoreServices.framework
- CoreGraphics.framework

- QuartzCore.framework
- libstdc++
- libz.dylib
- libxml2.dylib

5.1.2 Configuring the build settings

Open the build settings for each of your project targets that will link against the Agent's static library and add the linker flag `-ObjC` to the "*Other Linker Flags*" section, as shown in Figure 3.

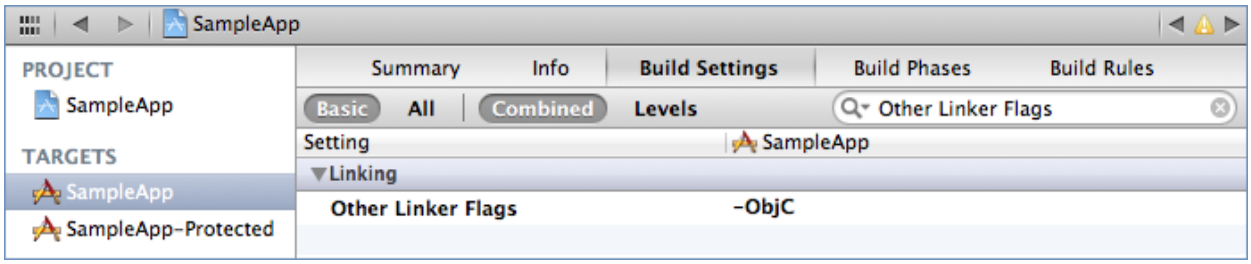

Figure 3 - Other Linker Flags

Make sure you add this flag to each of the build targets that will be linking against the static library.

You can now build your application with the development/unprotected version of the Agent and deploy it to a device or to the simulator. The Xcode debugger can be used without restrictions and the Agent will print quite some debug info to the console.

5.1.3 Creating a target to build the protected application

The Agent's protected library is built out of the exactly same codebase as the unprotected version but it implements all the anti-tampering measures enumerated in Section 3.2 on top of it. Due to this it requires a bit more of configuration work before attempting to use it in your project:

- A new special target must be created to build the application linked against the protected static library, which requires some specific build settings.
- The application's executable generated by Xcode needs to be patched for it to work properly with the protected library. This process is known as "finalization" and has to be performed as the last step of the build process, right after the linker has completed its work and before the code signing takes place. In order to fully automate this process, a new build phase has to be added to the build target, which will execute a shell script that will perform the finalization automatically and transparently for you.

The anti-tampering protection technology is not available on the iOS Simulator platform.

a) **Creating the new build target**

Select the target used to build your application and create a duplicate by right clicking on it and selecting "*Duplicate*" in the popup menu. Rename this target to something meaningful, for instance by adding the

"-Protected" suffix to its name.

Open the build settings for this new target and switch to the *General* tab. Remove the dependency of the unprotected Agent library from the list by selecting it and clicking over the *minus* button.

b) **Adding the protected library as dependency of the new target**

Locate the following 3 files under the `libDownloadableAgent/protected-lib` folder of the SDK package:

- `libDownloadableAgent-prot.a`
- `finalize1.fin`
- `finalize2.fin`

Copy them to the folder previously created and repeat the procedure described in 5.1.1 to add these files to Xcode. Now make sure that you **only select the new protected target** you have just created.
The final list of linked libraries should look as in the **Error! Reference source not found.**.

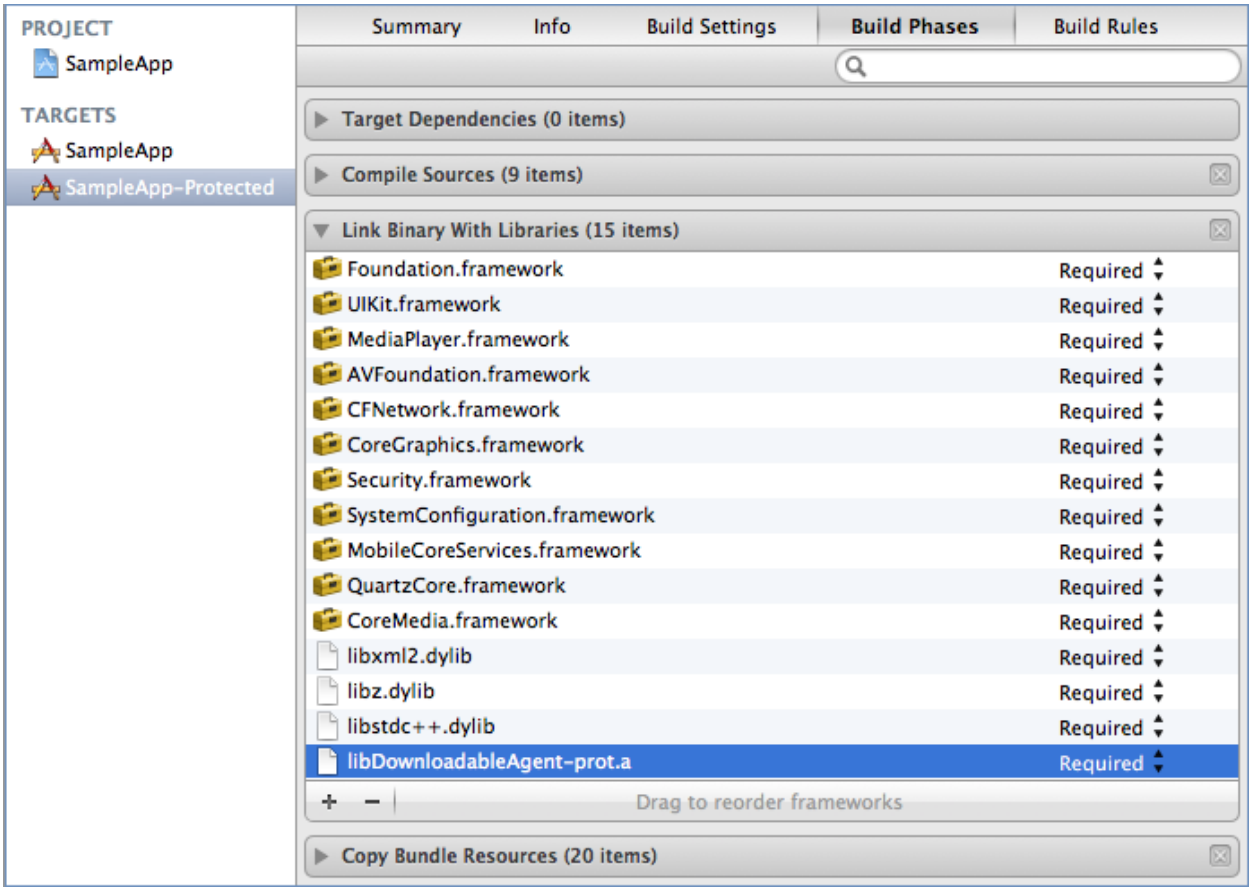

Figure 4 - Linked libraries for protected target

c) **Configure the new target**

Open the build settings for the protected target and adjust the value of the following settings to match the values on the right:

Setting	Value
Architectures	Standard (armv7)
Dead Code Stripping	disabled
Write Link Map File	enabled

d) **Add the build phase script to finalize the executable**

Locate the folder `finalization-tools` in the release package. Copy the entire folder somewhere into your project's root folder, for instance within the previously created folder to place the Agent libraries. Next, open up the file named `finalization_script.sh` with any text editor, select all its content and copy it to the clipboard.

In Xcode, click in the project's name on top of the left side panel and select the "*Build Phases*" tab. Find the "*Add Build Phase*" button on the bottom right corner and select "*Add Run Script*" in the pop up menu. Paste the script copied before into the numbered text box of the new build phase and let the default shell `/bin/sh` as is.

For convenience you may rename the new build phase script to something more meaningful like "Finalize Protected App".

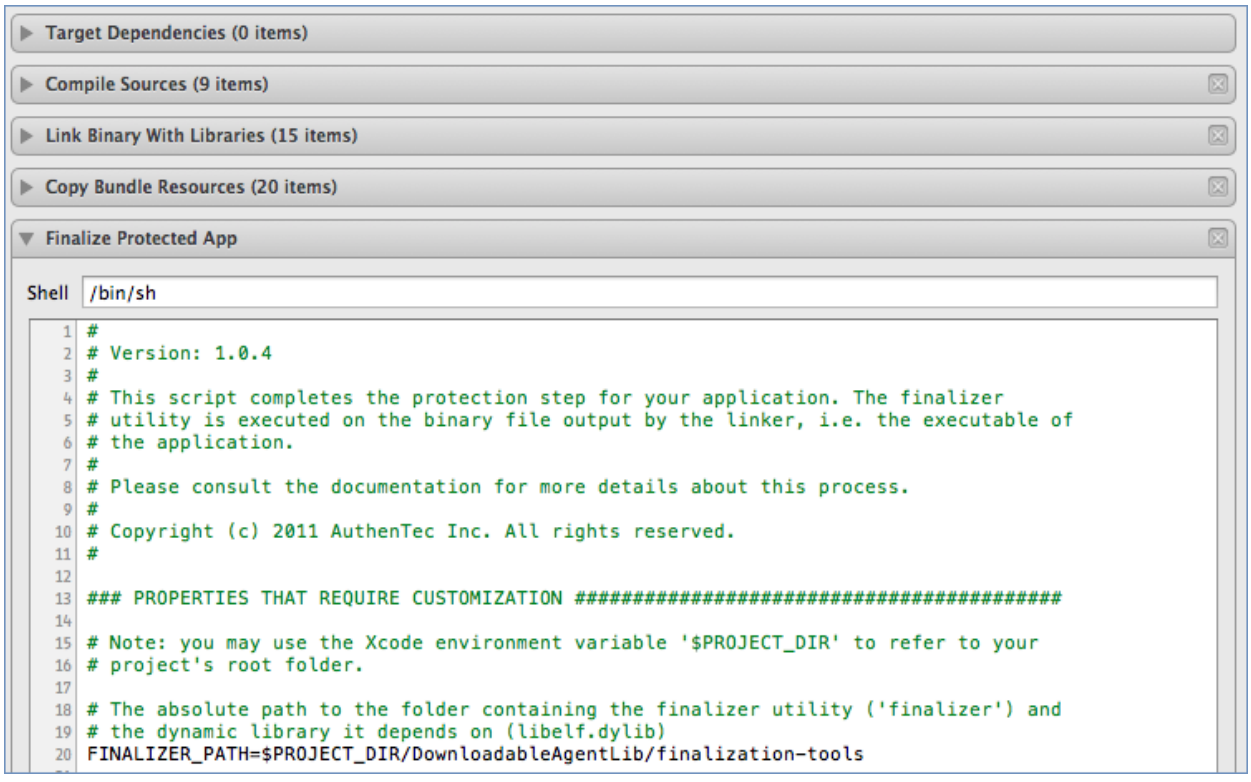

Figure 5 - The new build step for the protected target

e) **Configure the script**

Open the script again by double clicking on the new build phase and edit the value of following variables at the top:

- `FINALIZER_PATH`: Must contain the absolute path to the `finalization-tools` folder that you copied in the previous step. You may use the Xcode's `$PROJECT_DIR` environment variable as a pointer to the root folder of the project.
- `FINALIZATION_FILE_1` and `FINALIZATION_FILE_2`: These two variables must contain the absolute path to each of the two `.fin` files copied in the step b). Again, you may use the `$PROJECT_DIR` variable to avoid having to write the entire local path.

You should now be able to successfully build your application with the protected library and deploy it to an iOS device. Note that you should not attempt to run it from Xcode though, given that the anti-debugging protection will cause the application to crash as soon as any of the Agent methods is called while the GDB debugger is attached. Logging is also completely disabled in this version for security reasons.

5.2 Provisioning PlayReady Model Certificates

The first step before using the API is to provision the Agent with the device model certificates and their corresponding private keys. This is required in order for the Agent to securely communicate with PlayReady license servers.

5.2.1 Obtaining the PlayReady model certificates

The instructions to obtain your own company's PlayReady certificates from Microsoft are detailed in a separate document included in the *documentation* folder of the release package. Please see the corresponding PDF and follow all the steps carefully before proceeding to the next section.

5.2.2 Importing the certificates and keys into the Agent

The Agent provides the following static methods to import the encrypted certificates and private keys generated in the previous step:

```
+ (BOOL)importDeviceModelCertificate:(NSData *)certificate
        forModelCertificate:(ModelCertificateType)type;

+ (BOOL)importDeviceModelPrivateKey:(NSData *)privateKey
        forModelCertificate:(ModelCertificateType)type;
```

The certificates should be imported only once after the application has been launched and before attempting to create an instance of the Agent.

Each of the above methods shall be invoked twice, once for the PlayReady OEM certificate and private key and again for the Windows Media DRM OEM certificate and private key (the order in which this happens is irrelevant).

The model certificates can be loaded up as resources from disk or embedded into the application's source code as hex-encoded byte arrays, for instance. The reference implementation included in the release package follows the former method. See the *SampleAppAppDelegate.m* class for more details.

The reference implementation ships with Microsoft test certificates which must not be used in the final product.

All the invocations to the above methods should return a `TRUE` value, otherwise the whole certificate provisioning procedure should be considered as failed and all attempts to use the Agent are likely to fail as well, although this should be a very uncommon situation (i.e. corrupted files or the user has tampered with the application in some way).

5.3 Playing Protected Content

Playing DRM protected content with the Agent involves a series of steps that will be described in detail in the following sections.

The process in general is quite simple, as the Figure 6 illustrates. The flow starts when the Agent is initialized with a particular protected content, and then a license is acquired (by contacting the License Server) and installed if no playback rights are already installed for this content. The native player can then be invoked and the playback begins.

The Section 5.4 of this document will cover how to handle other more specific and unusual cases, such as when the rights expire during playback.

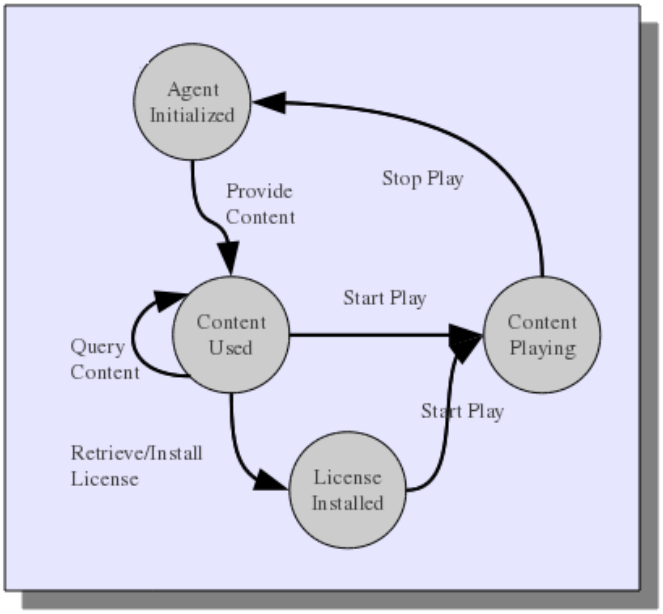

Figure 6 - Agent states during content playback

For a complete flow diagram of the messages and network calls between all the entities involved in the process of playing a protected content, please see the **Appendix B: Message flow diagram**.

5.3.1 Initializing the Agent

The Agent is initialized by allocating and initializing a new instance of the `DownloadableAgent` class. It provides a custom initialization method that must always be used: `-initWithContentURL:type:` which takes as parameters the URL of the content and the type of content expected (which MUST be one of the supported protocols). In the following example, the Agent is initialized to play a HTTP live streaming feed:

```
// the URL to a HTTP Live Streaming playlist
NSURL *contentURL = [NSURL URLWithString:@"http://example.com/movie/index.m3u8"];
DAContentType contentType = HTTPLiveStreamingType;

// create an instance of the Agent
DownloadableAgent *agent = [[DownloadableAgent alloc] initWithContentURL:contentURL
type:contentType];
```

The `contentURL` parameter must be a `NSURL` instance initialized with a valid URL. Should the URL be invalid, unreachable or point to a type of content or streaming protocol unsupported, the Agent will still be initialized successfully but will throw errors later on when checking the rights for the content.

See the following table to determine the correct input the Agent should expect for each type of content:

'contentType' parameter value	The 'contentURL' parameter is expected to point to
`HTTPLiveStreamingType`	A valid HLS playlist, either master or variant.
`SmoothStreamingType`	A valid Smooth Streaming client manifest.
`PIFFType`	An .ismv file, either stored on the local file system (the "*file://*" schema is allowed) or hosted in a remote web server.

The instance of the Agent obtained may only be used to manage the rights for the content it was initialized with, and it **MUST be retained for as long as the playback is in progress**. Once it has finished, the Agent can be safely released.

An instance of the Agent may be reused to play the same content repeatedly.

5.3.2 Rights and license information

The first step after initializing the Agent should always be to invoke `-contentInfo:error:` in order to determine the status of the content and to retrieve information about the currently installed rights (if any).

```
NSError *error;
DAContentInfo *contentInfo;

if (![agent contentInfo:&contentInfo error:&error])
{
    // the Agent failed to validate this content
    // examine and/or report the error code returned
    [...]
}
```

This operation may involve network activity and take some time to complete; therefore it should be executed on a background thread.

This method will only return a `TRUE` value when the Agent has been able to successfully inspect the file pointed by the content URL and it has passed certain validation steps. For instance, a valid HTTP Live Streaming playlist must, at the very least, start with the `#EXTM3U` tag and contain the URI to one or more transport stream segments or one or more variant playlists (which must also pass this validation). In the case of a Smooth Streaming client manifest, it must be a well formed XML document with at least one H264 video stream index (with one or more quality levels) and one AAC1 audio stream index with a single quality level.

If the operation fails it will return a `FALSE` value. In that case you may examine the specific error code returned in the `NSError` instance in order to consider whether the operation should be retried or not and if necessary display an error message. See the `DAErrorType` enumeration for a list of error codes returned by the Agent. The Agent always uses the constant `DownloadableAgentErrorDomain` as error domain.

Let's examine the properties of the `DAContentInfo` instance returned in order to determine how to proceed.

a) **Determining whether the content is protected with DRM or not**

The `isProtected` property indicates whether the content is protected with a DRM schema that the Agent can handle (currently only PlayReady DRM is supported). If the content is protected then you must always use the Agent to play it (i.e. you must obtain an instance of the media player by means of the Agent API). If it is not DRM protected, then use the Agent as well, with the only exception of unprotected HTTP Live Streaming content, which is natively supported by the player.

In summary, you may implement the following simple condition check:

- If the content is HTTP Live Streaming without DRM: Use Apple's frameworks to play it with the native player straight away. The Agent does not support playback of unencrypted HLS.
- In any other case: Use the Agent to instantiate the player and manage/acquire rights if necessary.

Only when the content is protected with a DRM schema you should proceed to check if there are any rights already installed for it before attempting to play the content.

b) **Determining the already installed rights**

The `rightsStatus` property indicates what playback rights are currently available for the protected content. With this information you can decide whether it is possible to proceed to play the content directly or it is necessary to acquire a license first.

The following table lists all the available rights:

Rights	Possible Action	Description
Valid	Play content	There are valid rights available for the content
No Rights	Obtain license	There are no valid rights available for the content
Rights in the Future	Obtain license	There are some rights available for the content but they are not valid yet
Rights Expired	Obtain license	There are some rights available for the content but they are expired
Untrusted Time	Reset system clock	Indicates that the system clock has been tampered with and cannot be trusted for data/time based rights.

A simple switch statement that covers each of the enumerated values is the easiest way to direct the application's flow depending on the current rights status as in the following example:

```
switch (contentInfo.rightsStatus)
{
    case DAValid:
    {
        // installed rights allow playback at the current date & time
        [...]

        break;
    }
    case DANoRights:
    case DAExpired:
    {
        // no rights are currently installed
        [...]

        break;
    }
    case DARightsInFuture:
    {
        // rights are installed but playback is not yet allowed
        [...]

        break;
    }
    case DAUntrustedTime:
    {
        // rights may be installed but cannot be used because the
        // local system time is not trusted
        [...]

        break;
    }
}
```

The possible values are quite self-explanatory, except perhaps on what should be a rare case of the rights status reporting a `DAUntrustedTime` value. This means that the Agent has detected a rollback in the system clock and cannot use the license installed for this content until the user adjusts the clock to the correct date. See section 5.4.2 for more information about this subject.

c) **Determining the license's life time**

When the rights status has a `DAValid` value, the installed license may or may not have a start date or/and an expiration date set. If it does have any of the two or both, then the Agent will populate the corresponding properties in the `DAContentInfo` instance with UTC based dates, otherwise those properties will be left with a `nil` value, which means that the content can be played without time restrictions.

If the rights status is `DARightsInFuture` instead, then the installed license has at least a start date defined, which is ahead of the device's local time (therefore it does not allow playback yet), and it may or may not have an expiration date. Again, the Agent will populate the corresponding properties in the `DAContentInfo` instance with UTC based dates.

In any case the dates defined in a license are expressed as absolute values based on the License Server's time. This means that it is **very important** that the client (the Device) and the server have accurate date & time values in order for the Agent to properly apply these time based restrictions. If, for example, the device's clock were set to 1 day ahead of the server's clock and the server were issuing licenses valid for a period of 24 hours from the acquisition date, these licenses would be already expired as soon as they are installed on the device.

If your license server uses absolute date & time values to define the license play rights (i.e. the start and/or end date), it's often recommended to allow for a certain margin of error in case the device's clock is not perfectly in sync.

d) **Special case: Licenses that expire only after the first use**

PlayReady allows for a special type of play rights that do not have a start or end date but instead will expire after a certain time interval that only begins to tick the instant the rights are consumed for the first time, i.e. after the playback of the protected content has begun.

To tell this type of rights apart you may use the `rightsExpireAfterFirstUse` property of the `DAContentInfo` class. This boolean property will have a value of `true` only before the expiration timer has kicked in. After that, the property will have a value of `false` and the play rights will have an expiration date set.

Unfortunately, due to technical restrictions in the PlayReady SDK, it is impossible for the Agent to report the duration of the validity period before the rights have been consumed.

5.3.3 Acquiring licenses

Once it's been determined that a new license is needed in order to play a protected piece of content, the next step is to initiate the license acquisition process, which involves contacting a PlayReady license server to obtain it. This process must always be triggered manually; the Agent will never initiate a license acquisition by itself even if the content cannot be played because no play rights are available, although in such case it will signal the condition so that you can trigger the appropriate action.

The sequence diagram below shows a high-level overview of the license acquisition process:

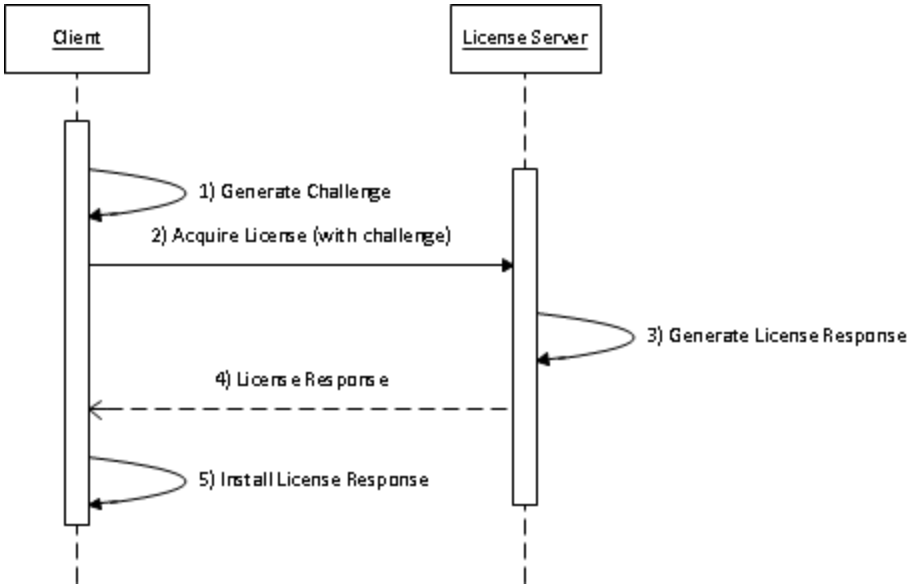

Figure 7 - Sequence diagram for a license acquisition

The Agent's API exposes a method to initiate the license acquisition process with the following signature:

```
- (BOOL)acquireLicenseWithDelegate:(id<LicenseAcquisitionDelegate>)delegate
                              error:(NSError **)outError;
```

This operation may involve network activity and thus take some time to complete; therefore it should never be executed on the main thread to avoid blocking the UI.

The Agent allows for this operation to be optionally performed with the help of a 'delegate' object, which when set will be *delegated* the task to send the license acquisition request to the server and pass the response back to the Agent.

The following sections describe both use cases, with and without a delegate, in detail:

a) **License acquisition without a delegate**

When the `delegate` parameter is set to `nil` the Agent will handle all the communications with the license server by itself and will try to process and install the response returned. If either the license request or the installation fails, the whole process is considered as failed and an error code is returned.

This operation is executed in a synchronous way by the Agent and cannot be cancelled. The boolean value returned by the method indicates whether the operation was successful or not.

b) **Delegated license acquisition**

The 'delegated' mode aims to offer more flexibility to the application integrator. In this mode, the delegate object is tasked with creating and sending the HTTP request to the license server, obtain the

response and pass it onto the Agent to be installed. Your delegate object may then, for instance, add custom HTTP headers or cookies to the request, or even route it through your own web portal in order to perform some sort of user authentication/validation first and then have your portal forward the request internally to the actual license server.

The delegate object passed as parameter must implement the `LicenseAcquisitionDelegate` protocol, which contains the following two methods:

```
@required
- (NSData *)performDelegatedLicenseAcquisition:(NSURL *)licenseAcquisitionURL
                                withChallenge:(NSData *)challenge;

@optional
- (NSData *)performDelegatedJoinDomain:(NSURL *)domainControllerURL
                                withChallenge:(NSData *)challenge;
```

During the delegated license acquisition process, the Agent will invoke the first of the methods passing a generated `challenge` and the `licenseAcquisitionURL`, which indicates where the request should be sent to. Note that the call to the delegate object is synchronous, so the delegate object will block the calling thread while the request is performed.

The second method is optional to implement in your delegate object and will only be invoked when a domain is required in order to acquire a license for a particular content. See the next section for more details about this use case.

The implementation of the delegate depends on your particular requirements and on how you want to communicate with your license server, but in summary these are the basic steps you'd need to perform:

1. Create an HTTP POST request with the URL provided. If you need to you may also add extra query parameters to it or even replace it with a different URL pointing to another intermediate server. The challenge, on the other hand, must not be modified as it is digitally signed.
2. Configure the request to talk a protocol your license server or portal can understand. PlayReady servers typically use the SOAP 1.1/1.2 protocol.
3. Set the challenge data as the body of the request.
4. Add any custom information to the request object, such as HTTP headers or cookies.
5. Send the request and obtain the server's response. Try to perform some basic error checking to tell between a proper response and an error message.
6. If the response body looks valid return it to the Agent, otherwise return `nil`.

For a working reference implementation of the `LicenseAcquisitionDelegate` protocol, please refer to the SampleApp project included in the release package (see the `SampleAppViewController` class).

Note that the successful acquisition and installation of a license does not necessarily guarantee that the content can be played right afterwards, as the license may contain rights that allow playback only in a future date. It is therefore recommended to always re-check the installed rights before attempting to start the playback, as in the following example:

```
if (licenseAcquired)
{
    // invoke [agent contentInfo:] again to ensure the license allows playback
    // at the current date & time
    [...]

    if (contentInfo.rightsStatus == DAValid)
    {
        // the playback can begin
        [...]
    }
    else
    {
        // the installed rights do not allow playback at this time!
        [...]
    }
}
```

5.3.3.1 Joining a domain when required

If your license server requires that the client device is on a domain in order to acquire a license for a particular content, the initial license acquisition request will fail with an error code that signals this condition. In this case the Agent will acknowledge it and try to initiate a 'join domain' operation.

If your delegate object implements the optional method declared in the `LicenseAcquisitionDelegate` protocol, it will be automatically invoked by the Agent and passed on the URL of the domain controller and a challenge, in a similar fashion to the license acquisition process.

Your delegate object is again responsible for composing and sending an HTTP request with the challenge provided and any extra data your domain server might require. Once you receive a valid response from the server, return it to the Agent to be processed and have the domain certificates extracted and saved.

If the join domain operation was successful, the Agent will retry the license acquisition process again by invoking for the last time your delegate with a newly generated challenge that contains the corresponding domain certificates.

See the sequence diagram below for the full list of calls between the different elements involved:

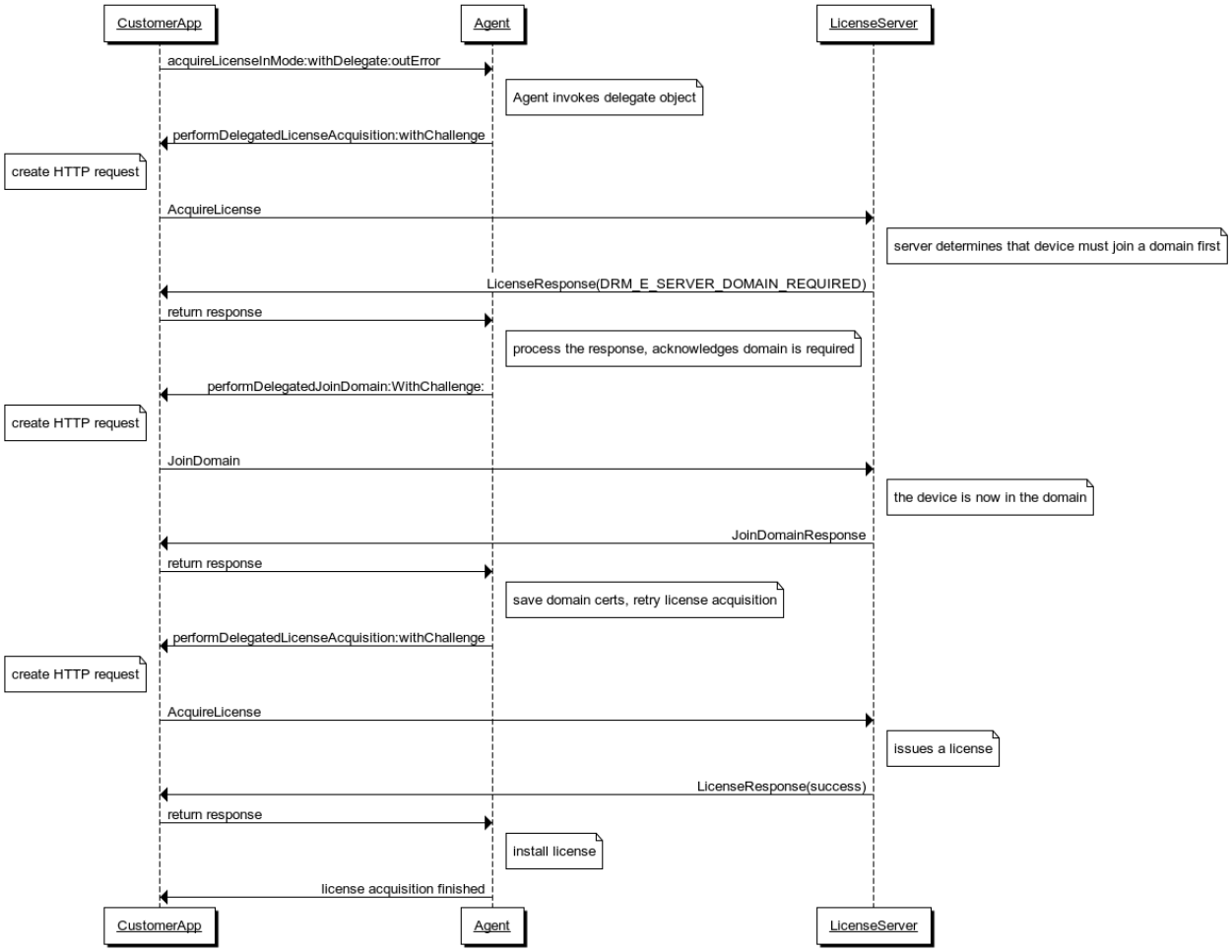

Figure 8 - Sequence diagram for a license acquisition with domain required

5.3.3.2 Adding custom data to the license acquisition challenge

If your license server is expecting you to send certain custom data in the challenge in order to issue a license for the content you may specify it **before** initiating the license acquisition process by using the method:

```
- (void)setCustomLicenseChallengeData:(NSData *)customData;
```

The custom data should be a piece of valid XML code. Note that the Agent does not interpret this data; it simply includes it in the license challenge message.

5.3.4 Setting restrictions for the HDMI output

The Agent enforces the Microsoft PlayReady Compliance and Robustness rules (http://www.microsoft.com/PlayReady/Licensing/compliance.mspx). Part of the compliance rules are

the OPLs (Output Protection Levels) that are present in the license and are determining if the content can be displayed on external outputs (like TV's, projectors, etc). In addition, the Agent provides an API that allows your application to explicitly override these rules when the HDMI output is being used. Note that the content decryption will not be possible if the license OPL requirements and/or HDMI control requirements are not fulfilled. Both these requirements will be referenced as "OPL check" or "OPL enforcement" in this section.

The HDMI client control API can be used on all iOS devices that have HDMI output. In case device doesn't have HDMI output, the HDMI control values will be ignored. With this API, the player application can explicitly set a policy to:

a) allow content display on external HDMI devices
b) not allow the external display
c) allow only audio
d) choose to display a predefined static image on the external display.

This API is based on two properties that you can set in any particular `DownloadableAgent` instance:

```
@property (assign) DAClientControlHdmiMode hdmiMode;
@property (retain) UIView* hdmiAlternativeImage;
```

The `hdmiMode` property specifies how the Agent will override the OPL enforcement rules built into the license (if any) when using the HDMI output for displaying or mirroring the audio/video content.

Possible values are:

Value	Description
`DAHdmiDefault`	The Agent will not override the built in OPL enforcement. OPL enforcement will be based on the OPL values from the license (if any). More information about the OPL levels and rules for their enforcement can be found in the Microsoft PlayReady Compliance and Robustness rules document.
`DAHdmiDisable`	The Agent will fail the OPL check in case the HDMI output is detected, even if the license OPL levels are allowing it. Failing the OPL check means that the content decryption will fail and therefore the playback will be aborted.
`DAHdmiAllow`	The Agent will allow the HDMI output, even if the license OPL levels are not allowing it.
`DAHdmiAudioOnly`	Only the audio part will be allowed via HDMI cable.
`DAHdmiUseAlternative`	In this mode, you must provide a UIView subclass that will be displayed on the external HDMI screen. The `hdmiAlternativeImage` property should be used for this purpose. This view will be added on top of the view hierarchy of the external screen's `UIWindow`, so make sure to set a proper auto-resizing mask on it.

The HDMI control will be enforced only during the playback of a protected content. It will never be enforced for clear text content. Also the values set on a particular instance of the Agent will only persist for as long as that instance is alive and will not be used on any other instances unless you explicitly set them. This way you may decide to enforce different rules on a per content basis.

In case the player sets the `hdmiMode` to `DAHdmiUseAlternative`, the `hdmiAlternativeImage` property should be set as well; otherwise the Agent will simply display a white screen on the HDMI output.

5.3.5 Selecting a particular audio track

If the content you are about to play contains multiple audio tracks it is possible to instruct the Agent to use one of them in particular. Otherwise the Agent will always default to the first audio track found.

This feature is currently supported **only for Smooth Streaming** streams.

In order to select an audio track you need to know the name of that track beforehand. Otherwise you may use the `contentInfo` method in the Agent's API to retrieve a list of all the audio track names available for the content. The `DAContentInfo` class has a property, `audioTrackNames`, which is an array of strings, each one representing the unique name of a particular audio track.

Once you have the name of the desired audio track, set the following property in the corresponding Agent instance to force its use during playback:

```
@property (readwrite, retain) NSString *selectedAudioTrackName;
```

If this property is set before starting the player then the playback will begin with the selected audio track. If the value of this property is changed during playback then player might stall for a few seconds while it loads the new audio track.

5.3.6 Preparing to play the content

Once you have determined that the playback is currently allowed by the installed rights, the next step is to obtain an instance of the native player configured to play it. To do this you **must** use the API provided by the Agent, unless the content format and protection schema are natively supported by the iOS media player, such as in the case of HLS streams without PlayReady DRM.

The following methods are available for this purpose:

```
- (MPMoviePlayerViewController *)moviePlayerViewController;
- (MPMoviePlayerController *)moviePlayerController;
- (AVPlayerItem *)avPlayerItem;
```

The first two methods return an instance of the player using the MediaPlayer framework, therefore you'll obtain a `MPMoviePlayerController` instance (optionally embedded in a simple full-screen view controller, if you invoke the `moviePlayerViewController` method). This player provides its own standard and widely known UI with some possibilities for customization, so it is a very convenient way to reproduce audio visual content. Please consult the iOS Reference Library page for more information.

The third method, `avPlayerItem`, returns an instance of the `AVPlayerItem` class, which uses the AVFoundation framework instead. This is an advanced Cocoa framework, a lower level abstraction than the MediaPlayer framework that allows you to customize a lot more how your player looks and behaves. Please consult the iOS Reference Library page for more information.

At this point, preparing to start the playback is no different than when playing non-DRM'ed content. Just create a canvas view for your player and instruct it to begin the playback the same way you'd do with a non-protected content. You should also subscribe to the notifications sent by the player in order to determine when the playback has ended and then proceed to release the instance of the Agent.

Important: the Agent instance must be retained until the playback is finished and the player has been dismissed.

5.3.7 During the playback

You don't really need to do much while the protected content is being played. The Agent will work transparently in the background to download, decrypt and deliver the media segments requested by the player in a secure way. Should a condition that requires your attention arise, the Agent will dispatch a notification and you application shall decide how to handle it. These notifications are covered in depth on Section 5.4.

There are other unpredictable issues however that can occur during the playback and that the Agent will try to handle by himself. To obtain a better understanding of its behavior, here is a brief description of what it does upon the most common situations:

- **Network timeouts or network unavailable**
  When the Agent encounters a timeout or the data connection is unavailable while trying to download a piece of content requested by the player, it'll try to force the player to re-request the same file internally. The player, however, may decide at any time to switch down to a lower bit-rate stream (if any is available) in order to avoid playback stalls when the network is not responding as fast as expected. This logic is completely under Apple's player control and cannot be modified by the Agent.
- **The content server reports an error**
  If any request to download a piece of content results in a response from the server which contains an HTTP error code (such as 404, 503, etc), the Agent will forward this very same error code back to the player. It is then up to the player to choose how to react to it and may vary between different iOS versions, but typically missing files result in that video segment or playlist being skipped, while server errors typically cause the player to terminate the playback.
- **Incorrect MIME type in the response from the content server**
  If a response from the content server reports an unexpected MIME-type, the Agent will do its best to try and parse the response as if it were the expected content. If it fails because the downloaded file does not have the expected format, it will then force the player to retry the request while at the same time it will fire up a notification to signal a possible issue with the internet connection (for instance, when all HTTP requests are being redirected to an HTML page, which typically happens when the device is connected to some Wi-Fi hotspots). See Section 5.4.3 for more information).

In all these cases error messages will be printed on the console to facilitate debugging and testing.

5.3.8 After playback ends

When the playback ends, typically after the end of the movie was reached or because the user tapped

the "Done" button on the player, it is recommended to release the Agent instance in order to free resources and memory, unless you intended to play the same content again right afterwards.

This is typically accomplished by subscribing to the `MPMoviePlayerPlaybackDidFinishNotification` notification and releasing both the player and the Agent in the method that handles it. This is the method used in the reference implementation's source code.

5.4 Handling Agent Notifications

While playing a protected content, the Agent may send notifications to inform about the occurrence of certain events that might require action. These notifications are broadcasted through the iOS Notification Center and are available to any observer object that has previously subscribed to them. See Apple's documentation for more information about the notifications system architecture.

The following sections describe in detail each notification sent by the Agent. Although is not strictly mandatory to implement handler methods for all of them, the playback will silently fail at some point if you choose to ignore these notifications.

5.4.1 Notification: Rights expired during playback

If the rights for a protected content expire during the playback, the Agent will broadcast a notification (`NSNotification`) whose name is defined as the constant:

```
extern NSString *const DownloadableAgentPlaybackRightsNotAvailableNotification;
```

Your application should subscribe to this notification before starting the playback by adding an observer object that implements a handler method, as in the following example:

```
[[NSNotificationCenter defaultCenter]
                 addObserver:self
                    selector:@selector(playbackRightsNotAvailable:)
                        name:DownloadableAgentPlaybackRightsNotAvailableNotification
                      object:nil];

[...]

- (void)playbackRightsNotAvailable:(NSNotification *)aNotification
{
    // renew the license for the content being played
    [self performSelectorInBackground:@selector(renewExpiredLicense)
                           withObject:nil];
}
```

Note that the object posting the notification is not necessarily your instance of the Agent, so you should always leave the `object` parameter with a `nil` value.

When this notification is fired it means that the Agent will no longer be able to decrypt the video stream until a new and valid license for that content has been installed, but it will hold off the player waiting for

that to happen. The responsibility to acquire and install new license is entirely yours, and if you fail to do so the player will eventually error out.

Given that the player typically keeps a hefty buffer of downloaded content, it should be able to continue playing while the license acquisition is performed on background, resulting in a completely transparent process with no visible interruption for the user. However, if the acquisition is delayed for too long the player may choose to time out on the current stream (because it is not receiving data) and switch to a lower or even the lowest bit-rate available in the playlist.

Optionally you can choose to programmatically pause the player when the notification is received in order to inform the user and/or ask for permission before renewing the license.

Frequency: This notification is sent only once when the rights for the content being played have expired and will not be sent again until after a valid license has been installed successfully.

5.4.2 Notification: Untrusted system time

The Agent implements the PlayReady Secure Clock feature, which is able to detect when the system clock has been manipulated (rolled back) in order to prevent the user from consuming the same date-based rights repeatedly by setting back the date and time on the device.

When a rollback is detected, all the date-based licenses that were already installed are disabled by the Agent, and any date-based license that is installed while in rollback mode will be immediately disabled as well (acquisition and installation will still succeed though). This restriction will remain in place until the local time is adjusted by the user and verified to be correct by the Agent, who will contact Microsoft's Time Service to obtain a trusted date. This will happen automatically every time the Agent is used (for instance, when retrieving the installed rights for a content). Note that this verification requires of a working data connection to succeed.

The Agent will never disable those licenses that do not have date/time based rights (i.e. no expiration date is defined in the license).

If a rollback is detected during the playback of content with a date-based license, the Agent will broadcast a notification (`NSNotification`) whose name is defined as the constant:

```
extern NSString *const DownloadableAgentUntrustedSystemTimeNotification;
```

Your application should subscribe to this notification before starting the playback by adding an observer object that implements a handler method, as in the following example:

```
[[NSNotificationCenter defaultCenter]
                 addObserver:self
                    selector:@selector(systemTimeNotTrusted:)
                        name:DownloadableAgentUntrustedSystemTimeNotification
                      object:nil];

[...]

- (void)systemTimeNotTrusted:(NSNotification *)aNotification
{
    // notify the user with a modal alert
    [self alert:@"A rollback to the system clock has been detected. Please correct
the date & time in order to continue playing the content"];

    // stop the player
    [...]
}
```

Note that the object posting the notification is not necessarily your instance of the Agent, so you should always leave the `object` parameter with a `nil` value.

The recommended actions upon receiving this notification are:

- Notify the user and suggest correcting the local date & time: On iOS devices it is always recommended to leave the "Set Automatically" option enabled in the Settings panel, in order to get automatic time synchronization from the network.
- Stop the playback completely: The Agent cannot recover from this condition until the clock is re-adjusted, so it typically makes sense to dismiss the player and wait for the user to take the appropriate action.

The Agent currently does not send any notification when the restrictions are lifted.

Frequency: This notification is sent only once when the issue is detected by the Agent and will not be sent again until after at least one piece of content has been decrypted successfully (i.e. the clock has been readjusted and validated by the Agent).

5.4.3 Notification: Invalid content downloaded

Should the Agent encounter a situation where a file requested by the player (for instance, a video segment) could not be parsed correctly, it will fire up a notification with the following name:

```
extern NSString *const DownloadableAgentDidDownloadInvalidContentNotification;
```

This could mean a number of things, but the most common cause is that the device has roamed into a Wi-Fi hotspot which requires some sort of user authentication or challenge in order to allow Internet access, and therefore the HTTP requests made by the Agent are being redirected and an HTML page is returned instead.

In this case it is recommended to alert the user to verify that the wireless data connection is working properly before resuming the playback. It is not mandatory to take action, but the player will meanwhile

keep trying to obtain the file repeatedly until the response is valid content. This could also result in the downgrade of the stream being played to the lowest quality stream after a while. Once the device has a working data connection, the playback should resume automatically.

Your application should subscribe to this notification before starting the playback by adding an observer object that implements a handler method, as in the following example:

```
[[NSNotificationCenter defaultCenter]
                 addObserver:self
                    selector:@selector(invalidContentDownloaded:)
                        name:DownloadableAgentDidDownloadInvalidContentNotification
                      object:nil];

[...]

- (void)invalidContentDownloaded:(NSNotification *)aNotification
{
    // notify the user
    [self alert:@"Your Internet connection appears not to be working properly.
Please verify and resume the playback"];

    // pause the player
    [...]
}
```

Note that pausing the player is not strictly required but consider that the user might need to quit your application in order to troubleshoot the data connection.

Frequency: This notification is sent only once when an invalid content is detected and will not be sent again until after at least one requested piece of content has been parsed successfully within the same playback session.

5.4.4 Notification: Server error received

This notification is fired when any of the HTTP request made by the Agent to retrieve a piece of content has failed with an HTTP error code (i.e. 4xx and 5xx codes) returned by the server. The purpose is just to help you identify potential issues within your content distribution network; therefore you should always log and report this particular notification for further investigation.

The notification's name is defined as the constant:

```
extern NSString *const DownloadableAgentDidReceiveServerErrorNotification;
```

The *user info* dictionary of this notification contains two objects:

- A `NSString` object that indicates the URL of the failed request
- A `NSNumber` object that indicates the HTTP status code returned by the server

The keys to retrieve these values are the following (respectively):

```
extern NSString *const DownloadableAgentDidReceiveServerErrorURLUserInfoKey;
extern NSString *const DownloadableAgentDidReceiveServerErrorCodeUserInfoKey;
```

Your application should subscribe to this notification before starting the playback by adding an observer object that implements a handler method, as in the following example:

```
[[NSNotificationCenter defaultCenter]
                addObserver:self
                   selector:@selector(receivedServerError:)
                       name:DownloadableAgentDidReceiveServerErrorNotification
                     object:nil];

[...]

- (void)receivedServerError:(NSNotification *)aNotification
{
    // Log the failed URL and the error code received

    NSString *failedURL = (NSString *) [[aNotification userInfo] objectForKey:
DownloadableAgentDidReceiveServerErrorURLUserInfoKey];
    NSNumber *errorCode = (NSNumber *) [[aNotification userInfo] objectForKey:
DownloadableAgentDidReceiveServerErrorCodeUserInfoKey];

    NSLog(@"A request to download the file with URL '%@' failed with the error code
%i", failedURL, [errorCode intValue]);
}
```

Note that since the Agent typically passes the result of each request back to the player, the later may decide to terminate the playback after receiving certain server errors. In particular, the 5xx type of errors often seems to cause the player to quit.

5.4.5 Notification: Playback should be aborted

This notification signals that something has failed internally during the playback and the Agent has determined that it might not be able to successfully recover from it; therefore the player might stall or enter into an error loop if it is not terminated. In this particular case it is recommended to force the player to quit gracefully.

The notification's name is defined as follows:

```
extern NSString *const DownloadableAgentPlaybackShouldBeAbortedNotification;
```

Your application should subscribe to this notification before starting the playback by adding an observer object that implements a handler method, as in the following example:

```
[[NSNotificationCenter defaultCenter]
                 addObserver:self
                    selector:@selector(shouldAbortPlayback:)
                        name:DownloadableAgentPlaybackShouldBeAbortedNotification
                      object:nil];

[...]

- (void)shouldAbortPlayback:(NSNotification *)aNotification
{
    // notify the user
    [self alert:@"Content playback has failed. Please report this error"];

    // log and report the issue internally
    [...]

    // stop the player
    [...]
}
```

It's difficult to predict beforehand what could cause this condition, but it should be considered as a very rare situation. Besides programming errors, other conditions that could result in this notification being fired are incorrectly protected content assets, which result in crypto errors. For instance, a piece of content that was encrypted with a key that no longer matches the encryption key delivered by the license server; or a content that has been partially encrypted with a different key(s).

In general is safe to say that this notification indicates a bug either in the product or in the content distribution and/or licensing network and therefore it should be reported for further investigation in order to determine the exact root of the issue. Make sure you log and submit as much info as possible (for instance, asset being played, type of device, software versions, etc) to help this task.

Frequency: This notification is sent as soon as the problem is detected, and might be sent multiple times during the same playback session if the player is not dismissed.

5.4.6 Notification: Playback will be aborted

Similar to the notification described in the previous section, this one signals that the Agent will proactively terminate the playback by interrupting the internal decrypted content feed to the player, which will eventually force the player to quit. It's therefore up to your application's code to make the player quit gracefully when the notification is received.

The notification's name is defined as follows:

```
extern NSString *const DownloadableAgentPlaybackWillBeAbortedNotification;
```

Your application should subscribe to this notification before starting the playback by adding an observer object that implements a handler method, as in the following example:

```
[[NSNotificationCenter defaultCenter]
                 addObserver:self
                    selector:@selector(willAbortPlayback:)
                        name:DownloadableAgentPlaybackWillBeAbortedNotification
                      object:nil];

[...]

- (void)willAbortPlayback:(NSNotification *)aNotification
{
    // notify the user
    [self alert:@"Content playback has been aborted. Please report this error"];

    // log and report the issue internally
    [...]

    // stop the player
    [...]
}
```

In this case however, the root cause is typically completely different to the previous case. The Agent will only decide to terminate the playback when there is a good reason for it, in other words, the current conditions do not allow protected content playback for either a technical or a security reason, such as when the platform's security has been compromised and is not considered secure enough to decrypt protected content (signs that an attacker is possibly trying to break or bypass the DRM security). The Agent implements several security checks that are executed at random intervals during the playback in order to try and detect these attacks.

Frequency: This notification is sent as soon as the problem is detected, and might be sent multiple times during the same playback session if the player is not dismissed.

5.5 Download & Play Offline

This feature allows you to progressively download and play a media file while it is also stored in a local cache that will allow playing it offline later on.

This feature is currently supported **for PIFF files only**.

The idea behind this is that the user is able to begin watching a certain online content that is streamed to the device and then resume watching it even when there is not any data connection available, such as for instance when the device is in "Airplane mode". If the content is not yet entirely cached then it shall be possible to play the arbitrary length of the content that is available in the cache.

The simple API provided by the Agent makes it really easy to accomplish this task, managing the entire download & caching process in the background and reporting the progress back to your application in real time. It's been designed to make your integration effort minimal.

The key for all content caching related operations is the `DACachedContent` class. It represents any content asset that is already stored or is to be stored in the cache. This class has been designed to abstract away most of the implementation details in order to make it simple and straight for you to operate. The only input required is, as usually, the URL of the content to be cached and a type that describes what the URL is pointing to.

5.5.1 Using the API to cache content

The `DACachedContent` class provides a static method that you **must** always use to obtain instances for each content item:

```
NSURL *contentURL = ...;
DAContentType contentType = PIFFType;

DACachedContent *cachedContent = [DACachedContent cachedContentWithURL:contentURL
ofType:PIFFType];
```

Once you have obtained an instance of this class you may use it to handle all the operations over the cache of that particular content. This instance can be disposed and re-obtained safely at any time invoking again the same method, given that its life cycle is managed internally by the Agent.

Use the following instance methods to control the caching process for a particular instance:

```
- (void)startCaching;
- (void)pauseCaching;
- (void)purgeCache;
```

All of these operations are performed asynchronously by the Agent, thus the methods above will dispatch them to an operation queue and then return immediately. Operations that are executing will continue to do so until after they are finished (successfully or after a repeated failure occurred) or you cancel them by invoking the `pauseCaching` method over the corresponding `DACachedContent` instance.

The limit as to how many operations can be run simultaneously is set by the operating system depending on the available resources and is not controlled by the Agent. Therefore you cannot assume that invoking `startCaching` for a certain instance will guarantee that the operation will begin immediately afterwards. In order to know when the operation has actually kicked off you must monitor the status of the cached content item, which is described in the next section.

Ongoing caching operations are suspended when the application enters into background mode and resumed when it is brought back to the foreground. However the Agent will not automatically resume these operations after a cold start of the application. You must restart them manually if so you wish.

Finally, should you want to obtain a list of all the content items that are currently stored in the cache, whether totally or partially, there is a class method just for that:

```
+ (NSArray *)allCachedContentItems;
```

This method will return an `NSArray` of `DACachedContent` objects, one for each content item with presence in the content cache. The `contentURL` property of each object will help you to identify each item.

5.5.2 Monitoring the progress and status of a cached content

The `DACachedContent` class exposes the following set of read-only properties that you may use to get the exact status of the caching process in real time:

Property	Description
`totalDuration`	The total duration of the content, in seconds, or -1 if the content is not yet in cache.
`cachedDuration`	The amount of content, counting from the beginning of it, that is currently stored in the cache and thus available for offline playback, in seconds. The value of this property will always be between 0 and [`totalDuration`].
`status`	The status of the caching process, one of: - Not cached - Caching in progress - Caching paused - Caching completed successfully - Caching failed See the `DACachingStatus` enum for the list of possible values.
`isReadyToPlay`	Whether or not a minimum amount of content has been cached that allows to start playback while in offline mode. You should not attempt to initiate the playback if the device is offline and this property has a value of false.

All the properties described above are KVO (Key Value Observing) compliant and thus can be observed to track any change in the progress and/or the status of the caching process.

For example, you could observe changes to the `cachedDuration` property in order to calculate the percentage of the total duration that is already stored in the cache to update a progress bar displayed to the user:

```
cachedContent = [DACachedContent cachedContentWithURL:contentURL ofType:contentType];

 // set ourselves as observers to monitor the amount of cached content
 [cachedContent addObserver:self forKeyPath:@"cachedDuration" options:(NSKeyValueObservingOptionNew)
context:nil];

...

// our KVO observer implementation
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change
context:(void *)context
{
    if ([keyPath isEqualToString:@"cachedDuration"])
    {
        DACachedContent *cachedContent = (DACachedContent *)object;
        float progress = cachedContent.cachedDuration / cachedContent.totalDuration;
        // update the progress bar in the main thread
        [self updateCachedProgressTo:progress];
    }
}
```

Remember that you must always tell the observed object(s) to remove your observers before proceeding to deallocate them; otherwise crashes are likely to occur.

5.5.3 Playing cached content

A content that is partially or fully stored in the cache can be played at any time. The procedure to play a cached content is exactly the same as for any other non-cached content, as the Agent will automatically fetch the segments from the cache if they are available.

However there are some considerations that should be taken into account:

- If the device is offline you might want to make sure that the user is not able to seek anywhere further than the timestamp defined by the `cachedDuration` property. This will ensure a pleasant user experience. Otherwise the player is likely to abort the playback due to the Agent being unable to fetch the requested segments when these are not available in the cache.
- On the other hand, if the device is online you may allow the user to seek anywhere within the movie length. In that case the Agent will attempt to fetch the missing fragments from the network and also store them in the cache **[*]**.

- You should always provide sufficient visual feedback during playback for the user to know what part of the movie is stored on disk (and thus very fast to seek into it) and what part is going to require the availability of a working data connection (slower response time).

We therefore recommend that you create a custom UI for the player where you can put the graphical elements and the logic necessary to implement such features/restrictions. In order to achieve this you will need to make use of the AVFoundation framework as opposed to the most commonly used MediaPlayer framework, which adds a bit more of complexity overall to your application but in exchange provides all the flexibility required for this particular use case. Please refer to Apple's documentation for more information on implementing a custom interface for the player.

One final consideration: it is allowed to play content while running any number of caching processes simultaneously in the background, even if a particular content is being both played and cached at the same time. However take into account that this will consume extra resources like CPU, memory and network bandwidth, which could result in degraded performance and consequently a worst user experience.

[*] Segments stored in the cache during playback will not appear reflected in the corresponding `DACachedContent` instance until after you resume the caching process (i.e. invoke the `startCaching` method) so that it can "catch up" with the new pieces of content added to the cache. This will be particularly true when you allow the user to seek anywhere further than the `cachedDuration` timestamp, as the `DACachedContent` only accounts for linearly cached content segments and thus it will have to "close the gap" between the last known cached segment and the new segments.

5.5.4 Purging the cache

Removing a cached content from the cache is as simple as invoking the `purgeCache` method shown in section 5.5.1. Invoking this method will dispatch an asynchronous command to stop the caching process -if it was running- and then proceed to delete all its cached data and metadata. The status of the corresponding `DACachedContent` will be then updated accordingly.

This operation may take a while to execute for large files. Please observe the `status` property to know with certainty when the operation has completed its execution.

Note that the cached content does not expire and it will not be cleaned up automatically by the Agent. It is therefore your responsibility to maintain the cache. In order to delete the entire cache (i.e. for all contents) you must obtain all the cached content items and then purge each one of them individually.

5.6 Debugging

The unprotected version of the Agent static libraries, for both Device and Simulator, do not restrict debugging in any way, therefore you can build & debug your application as you would normally with Xcode's GDB debugger. You should use these libraries during your normal development and testing cycle.

On the other hand, the protected library does include an anti-debugger feature, among other security measures, that will force the code to crash randomly when a debugger is detected. Therefore you should not attempt to run the protected target of your application while attached to Xcode.

Debug symbols are not available for any of the libraries.

5.7 Logging

A significant amount of logging is printed by the Agent to the Xcode Console in order to help debugging and testing the performance of the Agent and the player in general. Log sentences are classified in several levels that range from `DEBUG` to `ERROR`, but as of the current version it is not yet possible to define a desired threshold via the API.

Some examples of logged stuff are the following:

- The static library version, build number and build date.
- Details about license checks and acquisitions, such as all the files that have been downloaded internally in order to complete the operation, the existing rights installed, etc.
- Each request made by the player to the Agent's internal content proxy, including the time taken to download the requested file from the content server, the time taken to decrypt and deliver the content to the player and –depending on the content type- some more statistics, such as the network throughput.
- Issues detected during the playback, such as network timeouts, content server errors, incorrect MIME types reported, etc.
- Load and playback state changes in the player, such as when it is playing, seeking or paused.

The protected library does not produce any logging output for security purposes.

5.7.1 Untethered logging

Given that mobility is a "must" when it comes to test the performance of the Agent while playing streaming content on different networks, it usually comes very handy to have access to the Agent's logs and statistics while on the go and detached from the Xcode console.

In order to achieve this, the Agent uses the iOS Notification Center to make every log message produced available to your application through notifications (`NSNotification`) that carry these messages as payload. You can then, for instance, display such messages in a custom UI view in real time or write them to a text file for later inspection.

The name of these notifications is defined as the constant:

```
extern NSString *const DownloadableAgentDidPostLogMessageNotification;
```

You may subscribe and unsubscribe as a listener for these notifications at any time during the life time of you application. The reference implementation app included in the release package uses this method to display the logs in a separate view (often referred as “developer log” in the code).

6. Production Builds

When you are planning to build your application for publishing into the iTunes App Store, make sure that your application complies with the following:

- ✓ **Your production build is using the Agent's protected static library.**
  This library includes several protection technologies that increase its robustness against hacking attacks and grants the integrity of the DRM components.
- ✓ **Your application embeds production level PlayReady model certificates.**
  You must always use your own company certificates issued by Microsoft when creating a production build. Microsoft's test certificates, such as the ones included with our reference implementation app, shall only be used for test builds.
- ✓ **Your application's descriptor defines an ARMv7-only requirement.**
  The *Info.plist* file shall declare that the application is compiled only for the armv7 instruction set; otherwise your submission might be rejected by iTunes Connect. See the `UIRequiredDeviceCapabilities` property list key.

7. Known Issues

The following are known issues as of version 1.0 of the Downloadable Agent:

- **Protected content cannot be played when the 'HTTP Proxy' option is enabled in the Wi-Fi network configuration.**
  - Affects Platforms: All
  - Affects Firmware versions: 4.2.1 and newer
  - Description: This seems to be a bug introduced by Apple in the version 4.2.1 of iOS. When an HTTP Proxy is configured for the Wi-Fi network the device is connected to, the operating system will reroute all the HTTP traffic generated by 3rd party applications through the proxy, including the traffic addressed to the *localhost* interface, which appears to be a bug on Apple's code. Unfortunately this breaks the internal content proxy architecture that the Agent uses to feed content to the player, therefore playback won't work.

    This bug has been reported to Apple but to date there has been no response on when or if it will be fixed.
  - Solution: See **Appendix A** for instructions to detect when an HTTP proxy is being used.

- **Native player gets stuck when running the SampleApp on the iOS Simulator.**
  - Affects Platforms: iOS Simulator
  - Affects Firmware versions: all versions previous to iOS 5.0
  - Description: Trying to play any content, protected or unprotected, on the iOS Simulator often causes the player to completely freeze, forcing to restart the application in order to regain control. This is a bug in Apple's `MPMoviePlayerViewController` class that is only reproducible on the simulator platform, not on real devices.
  - Solution: Use the iOS 5 simulator, which is not affected by this issue. Otherwise use the `MPMoviePlayerController` class or the AVFoundation framework to play content instead.

- **Rebuilding a protected target without code changes on Xcode may result in a build error.**
  - Affects Platforms: iOS devices
  - Affects Firmware versions: All
  - Description: If you build a protected target twice or more times without making any changes in the source code, the Xcode build process will fail with an error. This is because Xcode does not rebuild the application's executable when no code changes were detected, yet this executable will still be submitted to the finalization service, and the finalization of an already *finalized* executable will always result in an error.
  - Solution: perform a "clean target" before invoking the build process again.

8. Upgrade Notes

This section highlights the changes in the API and other elements of the SDK that may require you to update your integration code when upgrading from a previous version.

8.1 Changes in version 1.1.0

This section contains instructions for customers upgrading their existing codebase to the Downloadable Agent for iOS version 1.1.0 (and newer).

Version 1.1 adds support for playing Smooth Streaming streams, whereas version 1.0 would only support HTTP Live Streaming feeds. It has therefore been necessary to introduce certain changes in the API, some to accommodate the new features and some other just in order to maintain a certain naming consistency. This means that the 1.1 libraries will not compile out-of-the-box when dropped into your 1.0-based project, but you'll need to so some manual changes.

- **File name changes**

  As you might have noticed already, the product's name has been changed in version 1.1. The HLSPlayReadyAgent has been renamed to the actual commercial name of the product, (DRM Fusion) Downloadable Agent. As a result of this, and since HLS is no longer the only streaming protocol supported, the public headers and libraries have also been renamed to keep consistency.

  The following table lists all the file name changes:

Previous name in version 1.0	Corresponding new name in version 1.1	Notes
HLSPlayReadyAgent.h	DownlodableAgent.h	
PRContentInfo.h	DAContentInfo.h	
libHLSPlayReadyAgent-dev.a	libDownloadableAgent.a	The new unprotected library is a universal library that contains binaries for both architectures.
libHLSPlayReadyAgent-sim.a	libDownloadableAgent.a	
libHLSPlayReadyAgent-dev-prot.a	libDownloadableAgent-prot.a	

- **API changes**

  The following classes, type defines and constants have been renamed:

Previous name in version 1.0	Corresponding new name in version 1.1	Notes
HLSPlayReadyAgent	DownlodableAgent	
PRContentInfo	DAContentInfo	
CI_ContentStatus	-	No longer exists on 1.1
CI_RightsStatus	DARightsStatus	

DAgentErrorType	DAErrorType	The enumerated error codes have also been refactored.
const HLSPlayReadyAgent*	const DownloadableAgent*	All constants previously defined with the 'HLSPlayReadyAgent' prefix have been renamed with the new 'DownloadableAgent' prefix.

The signature of the `DownlodableAgent` class initialization method has changed from:

```
- (id)initWithProtectedContent:(NSURL *)contentURL;
```

to:

```
- (id)initWithContentURL:(NSURL *)contentURL type:(DAContentType)contentType;
```

In order to accommodate the new parameter that specifies whether the content URL is HTTP Live Streaming or Smooth Streaming. This parameter is mandatory and must be known beforehand (i.e. you must know what kind of content you are initializing the Agent with).

The `PRContentInfo` (now `DAContentInfo`) class has also been slightly modified. The `contentStatus` property has been replaced by a simple Boolean property, `isProtected`, which provides the same functionality in an easier way. See the Section 5.3.2 for more information about how to use this property.

❖ **Other changes**

- ✓ You'll need to add a new iOS framework as dependency to your project: libxml2.
- ✓ The code of our reference implementation application (SampleApp) has been updated accordingly to integrate with the new version of the Agent. The content selection pop up has now content samples for both HTTP Live Streaming and Smooth Streaming, including protected and unprotected samples of each.

8.2 Changes in version 1.1.9

This section contains instructions for customers upgrading their existing codebase to the Downloadable Agent for iOS version 1.1.9 (and newer).

With the addition of support for PlayReady domains in version 1.1.9, the following changes apply:

❖ **API changes**

The declaration of the `LicenseAcquisitionDelegate` protocol has changed from:

```
@required
- (BOOL)performDelegatedLicenseAcquisition:(NSURL *)licenseAcquisitionURL
                                withChallenge:(NSData *)challenge;
```

To:

```
@required
- (NSData *)performDelegatedLicenseAcquisition:(NSURL *)licenseAcquisitionURL
                                withChallenge:(NSData *)challenge;

@optional
- (NSData *)performDelegatedJoinDomain:(NSURL *)domainControllerURL
                                withChallenge:(NSData *)challenge;
```

Besides the addition of the new optional method, the signature of the method that performs the delegated license acquisition has changed slightly to return an `NSData` object rather than a `BOOL` primitive. The delegate is now intended to return the server response data as the return value of the method instead of invoking the Agent to install the license via the `-(void)installLicense:` method, which has now been marked as deprecated to be removed in later versions.

The signature of the method to initiate the license acquisition has changed from:

```
- (BOOL)acquireLicenseInMode:(LicenseAcquisitionMode)mode
                withDelegate:(id<LicenseAcquisitionDelegate>)delegate
                       error:(NSError **)outError;
```

To:

```
- (BOOL)acquireLicenseWithDelegate:(id<LicenseAcquisitionDelegate>)delegate
                             error:(NSError **)outError;
```

Which means that the `mode` parameter has been removed in favor of using the sole presence of the delegate parameter to indicate whether or not to perform a delegated license acquisition. If the delegate is set to nil, the Agent will perform the whole process (a.k.a. Silent Mode).

8.3 Changes in version 1.2.4

This section contains instructions for customers upgrading their existing codebase to the Downloadable Agent for iOS version 1.2.4 (and newer).

Version 1.2.4 has simplified the usage of the API to play PIFF files, regardless of whether they are stored locally on the file system or remotely on an HTTP server. For this reason the `DAContentType` enumeration has had the types `FullyDownloadedPIFFType` and `ProgressivelyDownloadedPIFFType` merged into a single type valid for both use cases. Therefore the enumeration now contains the following elements:

```
typedef enum
{
   HTTPLiveStreamingType,
   SmoothStreamingType,
   PIFFType /** use for both local and remote PIFF files */
}
DAContentType;
```

Appendix A: Detecting the current HTTP proxy configuration

The following example code shows how to use Apple's CFNetwork framework to query the operating system about what HTTP proxy (if any) is currently configured for the data connection. This is useful in order to avoid trying to play content with the Agent unnecessarily when the system is affected by the HTTP proxy bug described in the **Known Issues** section.

```
- (BOOL)isAffectedByTheProxyBug
{
    // retrieve the current global proxy settings
    CFDictionaryRef proxySettings = CFNetworkCopySystemProxySettings();

    // ask the system if a proxy will be used for requests addressed to 'localhost'
using the HTTPS protocol
    NSURL *serverURL = [NSURL URLWithString:@"https://localhost"];
    CFArrayRef proxyList = CFNetworkCopyProxiesForURL((CFURLRef)serverURL,
proxySettings);

    // examine the first item returned
    CFDictionaryRef firstProxy = CFArrayGetValueAtIndex(proxyList, 0);

    BOOL proxyDetected = NO;

    if (CFDictionaryGetValue(firstProxy, kCFProxyTypeKey) != kCFProxyTypeNone)
    {
        // protected content cannot be played while a proxy is in use
        NSLog(@"protected content cannot be played currently");
        proxyDetected = YES;
    }

    CFRelease(proxySettings);
    CFRelease(proxyList);

    // unaffected iOS version or proxy not currently in use
    NSLog(@"HTTP proxy check successful: protected content can be played");

    return proxyDetected;
}
```

This code will always return `NO` on devices with a version of iOS older than 4.2.1, even when an HTTP proxy is in use, given that those versions of the operating system are not affected by the bug (i.e. do not forward loopback traffic to the proxy).

Note that in the Agent's architecture, the player will always be connecting to the internal content proxy through the loopback interface and using the HTTPS protocol, therefore we go further than just checking if a proxy is in use; we ask the system specifically if any proxy would be used when connecting to the URL *https://localhost/*. This ensures that we don't prevent playback on those systems where the HTTP proxy is set up via an auto-configuration script that excludes the loopback interface from the routing.

Appendix B: Message flow diagram

The diagram below represents a complete picture of the messages passed around between all the different entities that take part in the process of playing a protected content.

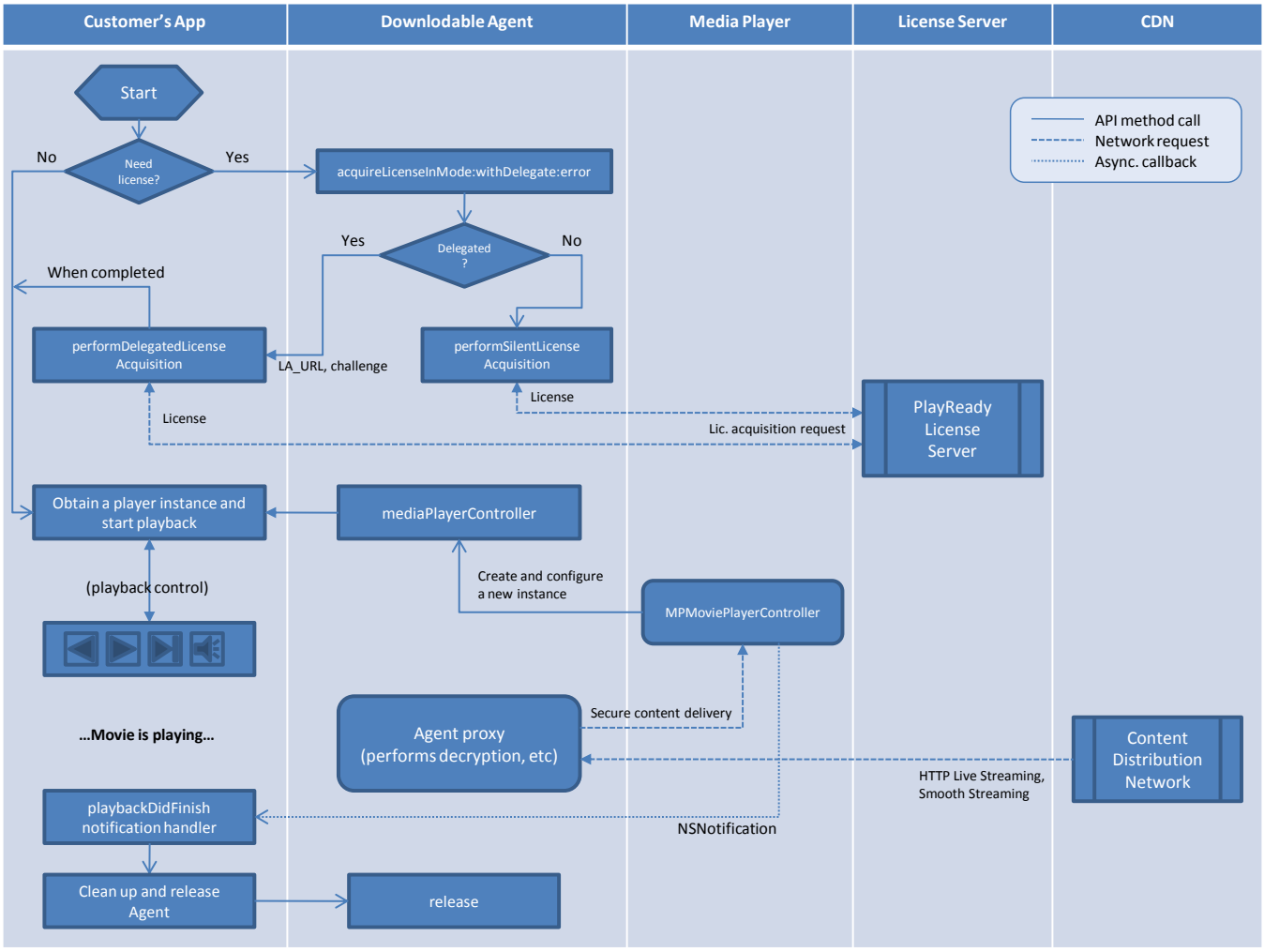

Figure 9 - Message flow diagram