TECHNICAL PAPER

Optimizing Video Startup Time
Adobe® Primetime DRM

By Eric Ha
Project Lead

February 2015

TABLE OF CONTENTS

INTRODUCTION

One of the most important factors in ensuring a satisfying user experience is the initial performance of the primary use case. For the video delivery ecosystem, this is quantified with the metric: "*After I hit PLAY, how long do I have to wait before my video starts?*" A near-instant start-up is the bar that we strive to attain. Noticeable waits result in unhappy customers; long waits result in lost customers and missed revenue.

Primetime DRM technology is for Digital Rights Management (DRM) of premium video content. In addition, the Primetime authentication solution, key to the TV-Everywhere initiative, leverages Primetime DRM technology, and can utilize most of the optimizations described in this paper.

DRM PLAYBACK STEPS

You must complete several steps before a client can play content protected by Primetime DRM. All of these steps require network communication. As a result, latency is observed, especially if the client has a high-latency network connection.

The steps to play Primetime DRM content are:

Step 1: Activate/Download the Primetime DRM Module (Flash Player Only)
Primetime DRM can be activated in Flash Player via the AS3 (ActionScript 3) API `flash.system.SystemUpdater.update()`. This call invokes a download of the DRM module for the current version of Flash Player. This module may be several megabytes in size. The device must complete this step before Flash Player can call any other Primetime DRM APIs.

Step 2: Individualize
After the DRM module has been downloaded to the device, a key provisioning process (called individualization) must run before the system can consume Primetime DRM content. The individualization process is automatically invoked on the first call to any Primetime DRM API (such as `flash.net.drm.DRMManager`). During individualization, the runtime communicates with the Adobe® Individualization server to set up a protected license store on the client device.

Step 3: Acquire a License (or Authenticate)
After the runtime has downloaded the Primetime DRM Module (Flash Player only) and completed the individualization step, the runtime can then handle Primetime DRM content. However, to play the content, the client must first acquire a playback license from a Primetime DRM license server. The license is acquired by sending a license request message to the appropriate license server. The server processes the request, performs any required authentication/authorization/entitlement, and returns a license that Flash Player can use to decrypt the video stream.

Step 4: Play Content
Once all previous steps have completed, you can initiate playback of the content via the `.play()` call.

All of the steps above must be completed before the device can play content. However, not all of them are necessary every time a new video is played. This is dependent on whether this is the first time the runtime has ever played protected content. The following scenarios indicate which of the above steps are necessary when playing protected content:

(1) Activate Primetime DRM (Download the DRM Module)

Flash Player must download the DRM Module when:

1) A client installs Flash Player for the first time
2) A client upgrades to a new version of Flash Player
3) Flash Player detects that the existing DRM Module has been tampered with/corrupted. This is always accompanied with a `DRMErrorEvent` 3343 error code when attempting to call into any Primetime DRM ActionScript API. In this situation, your ActionScript video player should respond to a 3343 by attempting to re-download the DRM Module.

(2) Individualize

The device must (automatically) individualize when:

1) (Flash Player) A new DRM Module has been downloaded.
2) (Flash Player) A new browser or computer user account is used to play Primetime DRM-protected content for the first time.
3) (iOS & Android) The application is installed or reinstalled.
4) (All Runtimes) The license store has been reset (e.g., by a `.resetDRMVouchers()` call, or a manual deletion of the license store, or a clearing of the web browser's settings).

(3) Acquire a License (or Authenticate)

The runtime must acquire a license from a Primetime DRM license server when:

1) `NetStream.play()` is called or a playback license cannot be found.
2) `DRMManager.loadVoucher("FORCE_REFRESH")` is called.
3) `DRMManager.loadVoucher("ALLOW_SERVER")` is called, and there is no license on the client.*

> **Note**: When `DRMManager.loadVoucher("ALLOW_SERVER")` is called, the device checks the local system to determine if a license is present. If not, it will then contact the license server to acquire a license.

While the Primetime DRM workflow involves a number of steps beyond the ones described above, the steps mentioned above are the most time-consuming, as information has to travel over a network connection. Once the steps are complete, a DRM decryption context is initialized on the client and is ready to begin decrypting protected video (or perform any other Primetime DRM operation, such as authentication). At this point, a call to `NetStream.play()` or `TVSDK.play()` on a Primetime DRM-protected video yields startup performance nearly equal to that of unencrypted video.

The worst-case scenario occurs when a client who has just installed (or updated) Flash Player attempts to play a protected video. The user must wait for a several-megabyte DRM Module to download, wait for client individualization to complete, and then wait for the device to request a license from the license server. Only at that point is video decrypted and presented onscreen.

Fortunately, there are several optimization options. Most of these options involve front-loading these time-expensive operations to earlier parts of your user's content consumption workflow. Because different users of Primetime DRM utilize different workflows and business rules, the techniques described here should be adapted to your particular situation.

OPTIMIZATION 1: FRONT-LOAD `SYSTEMUPDATER` API

Move the `SystemUpdater` ActionScript API as early in the workflow as possible, as this can be done independently and has no prerequisites. You might even consider putting this call at the beginning of the user experience workflow, when the user begins the process of purchasing the content.

After the client has successfully downloaded the DRM Module, it will not have to do it again unless the end-user upgrades their Flash Player.

For this approach to be successful, there are a few requirements that should be understood:

1. There is a SWF present on the page that the user is currently viewing.
2. The SWF stays loaded in the page long enough for the entire DRM Module download to complete.
3. Calling the `SystemUpdater` API when the DRM module is already downloaded results in a no-op, where a network call isn't even made.

Note #1
Call the `SystemUpdater` API as early as possible on a page where it's known that viewers remain for some time. You may even need to call `SystemUpdater` from a page that does not otherwise have any SWF assets. If this is the case, you can create an invisible SWF (or one with dimensions of 0 x 0) to embed into the page. From within this hidden SWF, the API can be called to download the DRM Module.

Note #2
As the call to download the DRM module is initiated from within a SWF, you must keep the SWF loaded on the current page until the download has completed. If the user navigates away, the current TCP/IP connections initiated from within the SWF will be destroyed as part of the normal SWF unloading process.

It is safe to call `SystemUpdater` repeatedly, as Flash Player gracefully handles DRM Module download failures. However, in an effort to conserve the end user's bandwidth, Adobe recommends that you embed the `SystemUpdater` call in a page the user is likely to view for a long time. One suggestion is the payment-entering workflow, or perhaps the HTML page behind the iFrame (if iFrames are used).

Note #3
In the case that a DRM module is present and valid, a subsequent call to `SystemUpdater` results in a NO-OP, and generates no network traffic.

OPTIMIZATION 2: FRONT-LOAD INDIVIDUALIZATION

Individualization can only be triggered by a call into a Primetime DRM API (such as any method of `DRMManager`). After individualization has happened once, the client will not have to perform the action again, unless their DRM system is reset.

In most workflows, individualization happens when a call is made to `DRMManager.loadVoucher()`, which acquires a license from the Primetime DRM license server. To leverage this side-effect, a DRM metadata of *any* protected content (which we will call a "dummy metadata") can be fed into this API in order to trigger an individualization. To further improve performance, the API can be called with: `.loadVoucher("LOCAL_ONLY")`. The use of `LOCAL_ONLY` instructs PrimetimeDRM to only check the local file system for a playback license (which will always fail if you use a dummy metadata). You'll immediately get a `DRMErrorEvent` indicating a license was not found, without having to wait for a network call to be made to the license server. As soon as the call fails/succeeds, your device has been individualized.

```
// Instantiate DRMContentData
DRMContentData cd = new DRMContentData( metadataByteArray )

// Instantiate DRMManager using DRMContentData
DRMManager manager = new DRMManager( cd )

// Induce individualization.  It is OK if the license acquisition fails here
// because we're only interested in entering any Primetime DRM workflow,
// which will automatically trigger an individualization behind the scenes.
Manager.loadVoucher( "LOCAL_ONLY" )
```

By doing this, the Primetime DRM can perform individualization at any time. In addition to getting individualization out of the way, this process will also initialize the underlying DRM Decryption Context, which performs the actual cryptographic operations when the time comes to consume and decrypt video.

OPTIMIZATION 3: `DRMMANAGER.INITIALIZE()`

For **ANDROID** and **IOS** Primetime, there is no need to use `DRMManager.loadVoucher()` in order to cause individualization as a side-effect. There is an API that was introduced which encapsulates the previous optimization of inducing individualization. In addition, other DRM-priming operations are performed for the purposes of improving video startup time when Primetime attempts to play a protected video.

A call to `DRMManager.initialize()` will invoke a license request using a pre-captured license request message that has been baked into the Primetime client, performing the individualization and initializing the DRM Context initialization described above.

When calling this API, it is not necessary to perform the previous optimization of "Front-Load Individualization", since this API encapsulates that logic.

OPTIMIZATION 4: PRELOAD LICENSE

Where possible, the video content server should make available the DRM metadata for its Primetime DRM content. FLV and F4V content have separate DRM metadata sidecar files during packaging. HLS, HDS, and DASH content contain the DRM metadata in their manifests.

DRM metadata is an artifact generated during the Primetime DRM packaging (encrypting) process. This metadata contains information needed by Primetime DRM to acquire and associate DRM licenses to protected content, such as the URL to the license server and the `licenseID` of the license needed to decrypt the content. All Primetime DRM content has the DRM metadata embedded into its header or manifest, which can be extracted by Primetime during playback. However, you can also generate external ("sidecar") metadata during content packaging.

If the DRM metadata is available to the device, Primetime has the ability to acquire a DRM license from the Primetime DRM license server ahead of time, before the user attempts playback of the video.

Some applications, especially adopters of Flash Media Rights Management Server (FMRMS), which predates Primetime DRM, tend to have the following inefficient workflow shown below. For simplicity, the sample is in ActionScript; the same logic applies to iOS and Android platforms.

```
// Play a video stream without determining if it is/isn't protected
Play( videoURL )
        // Register error and status event listeners
        registerEventListener( DRMErrorEvent, onDRMError )
        registerEventListener( DRMStatusEvent, onDRMStatus )

        // Play video stream
        netStream.play( videoURL )

onDRMErrorEvent( event )
        // Listen for an error event to indicate content is protected and not playable.
        // If protected, abort this workflow and acquire DRM license
        if ( event.ID  == "DRM.encryptedFLV" )
                DRMManager manager = new DRMManager( event.DRMContentData )

                // If license request is successful, a DRMStatus event will be dispatched
                Manager.loadVoucher()

                // Abort previous playback attempt
                netStream.stop()

onDRMStatusEvent ( event )
        // If the loadVoucher() call was successful, a DRMStatusEvent is dispatched.
        // Protected content now has a license and can be played
        netStream.play ( videoURL )
```

As an alternative to this multiple-callback-loop approach, Adobe recommends that you make the DRM metadata available, so that Primetime can be used to "pre-fetch" the license before making the initial call to "`play()`". In this way, the DRM Decryption context will be fully initialized and playback response will be comparable to the experience of playing unencrypted content.

Note: This workflow is currently not possible with Adobe® Open Source Media Framework (OSMF) v2.0. OSMF will automatically acquire a license when it detects DRM metadata within the stream or manifest. Support for pre-loading a license is not currently available. This optimization is only possible if the ActionScript application is not utilizing OSMF, or if OSMF is custom-modified by the developer.

OPTIMIZATION 5: NOT ENCRYPTING THE BEGINNING VIDEO

During the packaging process, Primetime DRM has the option of leaving the beginning portion of the video unencrypted. By leaving the beginning of the video unencrypted, the Primetime client can immediately begin video playback while the other DRM initialization and license acquisition steps occur in parallel. When the threshold of the unencrypted and encrypted boundary is reached, the client will seamlessly transition to the encrypted content.

In the Primetime DRM Java SDK, this feature is included in the Command Line tool packager "AdobePackager.jar" and is configured via the property file configuration:

```
encrypt.contents.secondsUnencrypted
```

If you are using another service/vendor/encoder/tool to perform your content packaging, please refer to their associated documentation on how to encrypt/package content so that the first several seconds of content is left unencrypted.

Below is an example of HLS content protected with Primetime DRM that has the segments 0-4 unencrypted, and the rest of the segments encrypted.

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-DISCONTINUITY
#EXT-X-KEY:METHOD=NONE
#EXTINF:10.00000,
fileSequence0.ts
#EXTINF:10.00000,
fileSequence1.ts
#EXTINF:10.00000,
fileSequence2.ts
#EXTINF:10.00000,
fileSequence3.ts
#EXTINF:10.00000,
fileSequence4.ts
#EXT-X-DISCONTINUITY
#EXT-X-FAXS-CM:……Bzixzf7oXU=
#EXT-X-KEY:METHOD=AES-
128,URI="http://faxs.adobe.com",IV=0x2c5601b56600960b8edc7a47e11e2a21
#EXTINF:10.00000,
fileSequence5.ts
#EXTINF:10.00000,
fileSequence6.ts
#EXTINF:10.00000,
fileSequence7.ts
#EXTINF:10.00000,
fileSequence8.ts
#EXTINF:10.00000,
fileSequence9.ts
#EXTINF:10.00000,
fileSequence10.ts
#EXT-X-ENDLIST
```

CASE STUDY

To illustrate the various optimizations described in this document, a hypothetical case study is presented below, along with Adobe's suggested best practices. Note that this is a typical, non-optimal implementation that lacks any of the optimizations covered in this document.

In the sample below, all of the logic for Primetime DRM playback is implemented within a single video player application. The entry point for all DRM related actions will be triggered **only** when the user clicks the "play" button, expecting to be presented with video. This is usually not optimal, and should be avoided. This code is written in ActionScript, but the same logic applies to all other Primetime clients.

The pseudocode for the **NOT-RECOMMENDED INEFFICIENT WORKFLOW** looks like this:

```
// Initialize handlers & play URL, not knowing if video is DRM-protected
playVideo( url )
{
        // handle errors that may require refreshing DRM Module
        stream.addEventListener( DRMErrorEvent, onDRMError )
        // listen for status event to determine if video is DRM-protected
        stream.addEventListener( StatusEvent, onStatusEvent )
        // listen for moment when DRM system is initialized & license is available
        stream.addEventListener( DRMStatusEvent, onDRMStatusEvent )

        // attempt to immediately play content, without knowing if it is DRM'd
        stream.play( url )
}

// Handle DRM Module update required
function onDRMError( event )
{
        // if Module is missing or corrupted
        if ( event.ID == "3344" || event.ID == "3343" )
        {
                // video cannot be played, so stop faulty playback
                stream.stop()

                // download new DRM module
                SystemUpdater.update( "DRM" )
        }
}

// Listen for status code to determine if video is DRM'd
function onStatusEvent( event )
{
        // if StatusEvent indicates video is encrypted
        if ( event.ID == "DRM.encryptedFLV" )
        {
                // video cannot be played, so stop faulty playback
                stream.stop()

                // extract DRMContentData from event
                drmContentData = ( event as StatusEvent ).DRMContentData

                // Initialize DRMManager & make asynch call to acquire license
                drmManager.loadVoucher( drmContentData )
        }
}

// Listen to determine if DRM has been initialized (and license is available)
function onDRMStatusEvent( event )
{
        // if DRMStatusEvent is returned, all DRM subsystems have been initialized,
        // license is available, and content is ready to be played.
        if ( event is DRMStatusEvent )
        {
                // video cannot be played, so stop faulty playback
                stream.stop()
```

```
                // play content via NetStream
                stream.play( url )
        }
}
```

This workflow should be avoided at all costs. Let's take a moment to reflect on what can be improved.

1. The code has no prior knowledge of whether the content is DRM protected or not. It is blindly playing the content and waiting for a `NetStatusEvent` to inform the video player if the content is protected and that the Primetime DRM workflow should be initiated. After the system is ready, the original `NetStream.play()` operation must be stopped and restarted. This is because playback of an encrypted stream when the DRM decryption context is not ready will fail and cannot be recovered.

Improvement: Revise the end-user workflow to inject information in advance so that it's known whether or not the content is protected.

2. The code waits for a `DRMErrorEvent` before determining whether or not it should invoke the `SystemUpdater` API to update the DRM Module on the device.

Improvement: Instead of waiting until the user clicks Play before determining if an error is dispatched (indicating a DRM Module download is required), take the initiative and call `SystemUpdater` ahead of time. Perhaps make this call when the user is typing their information for purchasing/renting the content.

3. The code waits for a `StatusEvent` to indicate that the content is protected. At that point, it extracts the `DRMContentData` from the event in order to initialize a `DRMManager` object. The `DRMManager` is then used to request a license from the license server. Typically, this workflow is used if there are business or technical reasons prohibiting the video content from being accompanied by the `DRMContentData` as a sidecar file. If there are no such restrictions in place, the Primetime DRM Java command line packaging tool (`AdobePackager.jar`) can be used to generate a `.metadata` file, along with the encrypted video. In addition, the Primetime DRM SDK can also be used to generate this sidecar file.

Improvement: Instead of waiting for an event to come back indicating that the content is protected and a license should be acquired, the application should inspect the content's metadata or manifest. This informs the player that the content is protected, and that the player should pre-download the license before playback is initiated.

SUGGESTED WORKFLOW

The following snippet demonstrates the suggested workflow using the optimizations described above.

```
// Initiate DRM Module download very early, perhaps at website landing page
updateModule()
{
        SystemUpdater.update()
}


// Acquire the license as soon as possible, ideally well before user initiates playback
acquireLicense( sidecarMetadata )
{
        // Register StatusEvent, DRMStatusEvent, and DRMErrorEvent handlers
        netstream.addEventListener( DRMErrorEvent, onDRMError )
        stream.addEventListener( StatusEvent, onStatusEvent )
        stream.addEventListener( DRMStatusEvent, onDRMStatusEvent )

        // Load metadata from sidecar file & create DRMContentData
        ByteArray ba = new ByteArray( sidecarData )
        DRMContentData cd = new DRMContentData( ba )

        // Create DRMManager and Acquire license
        DRMManager manager = new DRMManager( cd )
        manager.loadVoucher( "ALLOW_SERVER" )
}


// Play the video.  By now, all of the preconditions should be met and playback
// should immediately start
playVideo ( url )
{
        netstream.play( url )
}
```

ONLINE RESOURCES

Adobe® Access AS3 Developer Guide
http://help.adobe.com/en_US/as3/dev/WS5b3ccc516d4fbf351e63e3d118666ade46-7ce3.html

Adobe® Access ActrionScript Documentation
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/drm/package-detail.html

Adobe® Access Product Page & Documentation
http://www.adobe.com/support/documentation/en/flashaccess/index.html

ABOUT THE AUTHOR

As Project Lead for Primetime DRM Engineering, Eric has his hands in most of the bleeding-edge video delivery, protection, and monetization systems at Adobe Systems Inc. In a previous life, he may admit to having developed software systems for medical devices.