TECHNICAL PAPER

Optimizing "Start to Playback" performance with **Adobe® Access** content

By Eric Ha
Project Lead

October 2013

TABLE OF CONTENTS

INTRODUCTION

One of the most compelling factors of a satisfying a rich user experience is the initial performance when entering a use-case. For the video delivery ecosystem, this is most often quantified with the metric: "*How long do I wait to see the video after I hit PLAY?*" A rapid start-up is the bar that we, as developers, strive to attain. Long waits result in a poor experience and unhappy customers; even-longer waits result in lost customers and revenue.

The primary use of Adobe® Access technology is in Digital Rights Management (DRM) for premium video content. In addition, the Adobe® Pass solution, key to the TV-Everywhere initiative, leverages Adobe® Access technology, and can utilize most of the optimizations described in this paper.

DRM PLAYBACK STEPS

You must complete three steps before a client can play content protected with Adobe® Access DRM. All of them require network communication, and as such, consume a perceivable amount of time, especially if the client is connected to the Internet with less-than-ideal network latency and throughput.

Step 1: Activate/Download the Adobe® Access Module (Only needed for Flash Player)
On demand, Adobe® Access can be enabled in Flash Player via the AS3 (ActionScript3) API `flash.system.SystemUpdater.update()`. This call instructs Flash Player to download a DRM module for the current version of Flash player. This module can be several megabytes in size. The end-user system must complete this step before Flash Player can call any Adobe® Access APIs.

Step 2: Individualize
After the DRM module has been downloaded and installed on the system, a configuration process (called individualization) must run before the system can consume content protected using Adobe® Access. The individualization process is automatically invoked on the first call to any Adobe® Access related API (such as `flash.net.drm.DRMManager` or `NetStream.play()` on An Adobe® Access protected video). 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 Adobe® Access DRM Module (Flash Player only) and completed the individualization step, the runtime can now consume Adobe® Access protected content. However, to consume the content, the client requires a license to decrypt the content. The license is typically acquired by the runtime sending a license request message to an Adobe® Access license server. The server processes the request, performs any required authentication/authorization/entitlement, and returns a license that Flash Player can use to decrypt and render the video stream.

Playing Content
Once these 3 steps are completed, you can initiate playback of the content via a `.play()` call.

All of the steps described above must be completed before the client can view the content. However, not all of them are always necessary every time a video is played. In particular, this is dependent on if this is the first time this particular runtime *version* has ever played protected content. The following scenarios dictate which of the 3 steps above are necessary when playing protected content:

Activate Adobe® Access (Download a new 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/corrupted. This is always accompanied with a `DRMErrorEvent` 3343 error code when attempting to call into any Adobe® Access AS3 API. In this situation, your AS3 video player should respond to a 3343 error code by re-downloading the DRM Module.

Individualization

The runtime 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 Adobe® Access content for the first time.
3) (iOS & Android) The application is installed/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)

Acquire a License (or Authenticate)

The runtime must acquire a license from an Adobe® Access license server when:

1) `NetStream.play()` is called on a protected video and there is no license on the client.
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 Flash Player runtime checks the local system to determine if a license is present. If not, it will then (and only then) contact the license server to acquire a license over the network.

While the Adobe® Access workflow involves a number of steps beyond the ones described above, the steps mentioned here are the most time-consuming, as information has to flow over the (sometimes slow) Internet. For very slow Internet connections, this process can take up to 45 seconds, depending on the platform and Operating System. Once the steps are complete, the DRM decryption context on the client is initialized and ready to begin decrypting protected video (or perform any other Adobe® Access operation, such as authentication). At this point, a call to `NetStream.play()` on an Adobe® Access protected video yields start-to-playback performance nearly equal to that of unencrypted video.

The worst-case scenario occurs when a client who has just installed (or updated) their Flash Player attempts to play a protected video. The user must then 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 suggestions involve front-loading these time-expensive operations to earlier parts of your user's content consumption workflow. Because different users of Adobe® Access 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` AS3 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. The caveat with this approach is that this API can only be called from within a SWF, as it is an ActionScript 3 call.

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

For this approach to be successful, there are a few requirements 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 present results in a no-op.

Requirement #1
Call the SystemUpdater API as early as possible on a page where it's known that they will stay on 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.

Requirement #2
As the call to download the DRM module is initiated from within a SWF, you must keep the SWF loaded on the current HTML 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, it's recommended 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).

Requirement #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 an Adobe® Access API (such as `DRMManager`). After individualization has happened once, the client does not need to individualize again, until they upgrade to a new version of Flash Player.

Typically, this happens when a call is made to `DRMManager.loadVoucher()`, which acquires a license from the Adobe® Access license server. If desired, the DRM metadata of *any* protected content can be used, and the call specified as `.loadVoucher("LOCAL_ONLY")` can be used to ensure there is no extraneous network traffic generated. The result of this call (PASS or FAIL) and the source of the DRM metadata are irrelevant, as the machine will individualize even if no license can be acquired.

```
// 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 Adobe Access workflow,
// which will automatically trigger an individualization behind the scenes.
Manager.loadVoucher( "LOCAL_ONLY" )
```

By doing this, the Flash Player 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** implementations of the Primetime Media Player SDK, there is a single API which encapsulates the previous optimization of inducing an individualization by performing a (non-necessary) license acquisition ahead of time.

A call to DRMManager.initialize() will invoke a license request using a pre-captured license request blob, performing the individualization and DRM Context initialization described above. When calling this API, it is no longer necessary to perform the previous optimization of "Front Load Individualization", since this API encapsulates that logic.

OPTIMIZATION 4 – LOAD LICENSE OUT OF BAND

It is recommended that where possible, the video content server should make available the DRM metadata for its Adobe® Access protected content. The metadata is an artifact generated during the Adobe® Access packaging (encrypting) process. This metadata contains information needed by Flash Player to acquire and associate Adobe® Access licenses to a protected content, such as the URL to the Adobe® Access license server, and the `licenseID` of the license needed to decrypt the content. All Adobe® Access content has the DRM metadata embedded into its header, which can be extracted by Flash Player during playback. However, you can also generate external ("sidecar") metadata that can exist outside of the protected content.

A file containing sidecar DRM metadata is automatically generated and output to disk by the Adobe® Access Java packaging tools. If sidecar DRM metadata is available (which is used to create a `DRMContentData` object), the application has the ability to acquire the DRM license from the Adobe® Access license server at any time (*out-of-band*), rather than only during the beginning of media playback.

By having sidecar DRM metadata for content available, the application is not bound to first having the content stream available before it can request a license. Some applications, especially adopters of Flash Media Rights Management Server (FMRMS), which predates Adobe® Access, tend to have the following workflow:

```
// 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, it is recommended to make the DRM metadata available, so that the application can "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 AS3 application is not utilizing OSMF, or if OSMF is custom-modified by the developer.

OPTIMIZATION 5 – PARTIALLY UNENCRYPTED CONTENT

During the packaging process, the Adobe® Access Java SDK has the option of leaving the beginning portion of the video unencrypted. By leaving the beginning of the video unencrypted, the Flash Player 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 Flash Player will seamlessly transition to the encrypted content.

In the Adobe Access 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 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.

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 Adobe® Access DRM playback is implemented within a single video player application. As such, 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.

The pseudocode **FOR THE NOT-RECOMMENDED 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 if possible. 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 Adobe® Access 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.

Possible improvement: Revise 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.

Possible improvement: Instead of waiting until the user clicks Play before determining if a DRM Module download is required, do this earlier. Perhaps make this call when the user is typing data to purchase/rent 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 Adobe® Access Java command line packaging tool (`AdobePackager.jar`) can be used to generate a `.metadata` file, along with the encrypted video. In addition, the Adobe® Access SDK can be used to also generate this sidecar file.

Possible 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 have the content's metadata available, either created by the Java command line packaging tool or directly from the SDK. This informs the player that the content is protected, and that the player can pre-download the license before playback is initiated.

SUGGESTED WORKFLOW

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

```
// 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 a Project Lead, 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.