Adobe® Primetime

Video heartbeat SDK Guide for JavaScript - Version 1.5

Updated: 31 March, 2015

Contents

JavaScript Players

This guide describes how to add video heartbeat measurement to any video player that provides a JavaScript API. Most web-based players provide a JavaScript API, even if the underlying video is delivered in Flash or other video formats.

For example, the following players can be tracked using JavaScript:

- YouTube
- Brightcove
- Kaltura
- Ooyala
- HTML 5 (using native JavaScript support in the web browser and the *HTML 5 video events*).

For other players, implementing video heartbeat requires that your video player provides a JavaScript API with the following:

- An API to subscribe to player events. The video heartbeat SDK requires that you call a set of simple functions as actions occur in your player.
- An API or class that provides player information, such as video name and playhead location. The video heartbeat SDK requires that you implement an interface that returns current video information.

Requirements

Integrating video heartbeat requires the following:

- Existing Analytics implementation. These instructions assume that you have an existing implementation of AppMeasurement that is also using the Marketing Cloud Visitor ID Service. If you have not yet implemented Analytics or the Marketing Cloud Visitor ID Service, use the *Adobe Analytics Implementation Guide* and the *Marketing Cloud Visitor ID Service Guide* to get started.
- `VideoHeartbeat` library. (download instructions are in this guide)

Note: *Make sure your Analytics implementation is configured to send data to a development report suite before you start development.*

Example Implementations

An example is available in the `samples` folder that is included with the Video Heartbeat library.

Implementation Process

Complete the following steps to add video heartbeat tracking to your player:

Download the Video Heartbeat Library

Video heartbeat is distributed using a public Github repository.

1. Browse to *Adobe Github Video Heartbeat* and download the latest release for your platform.
2. Extract the zip, and copy `VideoHeartbeat.min.js` to a location accessible to your project. Optionally, copy the non-minified version to your project for debugging.
3. Save the `samples` folder to a location where the sample project can be reviewed and tested.

Next step: ***Configure AppMeasurement***

Configure AppMeasurement

The JavaScript implementation is configured similar to the ActionScript implementation on your website. The standard *Analytics Variables* are all available. Video Heartbeat also requires that you implement the *Marketing Cloud visitor ID service*.

1. Instantiate and configure the Marketing Cloud visitor ID service:

```
// Visitor

var visitor = new Visitor("INSERT-MCORG-ID-HERE");

visitor.trackingServer = "INSERT-TRACKING-SERVER-HERE";
```

2. Instantiate and configure AppMeasurement:

```
// AppMeasurement

var appMeasurement = new AppMeasurement();

appMeasurement.visitor = visitor;

appMeasurement.trackingServer = <tracking-server>;

appMeasurement.account = <rsid>;

// ... other AppMeasurement-specific configs (e.g., pageName, currency etc.)
```

   At a minimum, configure the following three variables:

   - `appMeasurement.account`
   - `appMeasurement.trackingServer`
   - `appMeasurement.visitor`

Next step: *Implement VideoPlayerPluginDelegate*.

Implement VideoPlayerPluginDelegate

The VideoPlayerPluginDelegate is used by video heartbeat to get information about the currently playing video, ad, and chapter.

Note: *This video player plugin delegate was previously named* `PlayerDelegate` *in version 1.4.*

First, read *How VideoPlayerPluginDelegate Works* to understand the role of `VideoPlayerPluginDelegate` in video heartbeat. Implementing this interface is where you will typically spend a majority of your implementation time.

To get started creating your own `VideoPlayerPluginDelegate` implementation, create a new object that uses `ADB.va.VideoPlayerPluginDelegate` as the object prototype:

```
var myDelegate = new VideoPlayerPluginDelegate();
```

Now that you have a player delegate, you need to define the functions that return information about your video and player:

```
function VideoPlayerPluginDelegate() {}

VideoPlayerPluginDelegate.prototype.getVideoInfo = function() {};

VideoPlayerPluginDelegate.prototype.getAdBreakInfo = function() {};

VideoPlayerPluginDelegate.prototype.getAdInfo = function() {};
```

```
VideoPlayerPluginDelegate.prototype.getChapterInfo = function() {};

VideoPlayerPluginDelegate.prototype.getQoSInfo = function() {};
```

Now that you have the framework in place, complete the rest of the sections in this doc to update these methods to return useful data from your player.

Video Information

The `getVideoInfo` method returns a `VideoInfo` object that contains details about the video player and the currently playing video. Before you can define this object, you'll need to use the API documentation provided by your player to find out how video information is retrieved. Video information is usually a property of the player object, or retrieved using a private method.

For example, In HTML 5, the playhead is a property of the `<video>` element:

```
document.getElementById('movie').currentTime;
```

In the YouTube API, the playhead is returned by a method call exposed by the player:

```
player.getCurrentTime();
```

To implement your custom `getVideoInfo` method, you'll need the following information:

Parameter	Required?	Description
`playerName`	Yes	The name of the video player that is playing back the main content
`id`	Yes	The ID of the video asset
`name`	No	The name of the video asset (opaque string value)
`length`	Yes	The duration (in seconds) of the video asset. If `streamType` is set to `vod`, return the length of the video. For other video types, return -1 as the length.
`playhead`	Yes	The current playhead location (in seconds) inside the video asset (excluding ad content) at the moment this method was called.
`streamType`	Yes	The type of the video asset.

After you have figured out how to get the required information, update the `getVideoInfo` method to return a `VideoInfo` object with the video information. How you populate each value is up to you, and varies based on your player. For example, you might load the video player name using a configuration file, or you could hard-code the value if you use only one player.

Ad Break Information

Ad breaks provide insight as to when a particular ad was displayed. For example, if you have a pre-roll and a midpoint ad break, you can collect position data along with the specific ad data. If you have only one ad break, you can simply provide 1 for the position and leave the name blank.

Parameter	Required?	Description
`playerName`	Yes	The name of the video player responsible with playing back the current advertisement break.
`name`	No	The name of the ad-break.
`position`	Yes	The position (index) of the pod inside the main content (starting with 1).
`startTime`	No	The offset of the ad-break inside the main content (in seconds). Defaults to the playhead inside the main content at the moment of the `trackAdStart` call.

Ad Information

Ad information is retrieved using a similar process used to retrieve video information, except you return an `AdInfo` object instead with details about the currently playing video ad. Use the API documentation provided by your Ad vendor to determine the following:

Parameter	Required?	Description
`id`	Yes	The ID of the ad asset
`length`	Yes	The duration (in seconds) of the ad asset
`position`	Yes	The position (index) of the ad inside the parent ad-break (starting with 1)
`name`	No	The name of the ad asset (opaque string value)

After you have figured out how to get the required information, update the `getAdInfo` method to return an `AdInfo` object with the ad information.

Chapter Information

If you are tracking chapters, you'll need to coordinate the chapter information returned with each call you make to `trackChapterStart`. Since chapters are likely defined by you and not your video player, you'll need a way to retrieve chapter definitions to populate this object.

Parameter	Required?	Description
`name`	No	The name of the chapter (opaque string value)
`length`	Yes	The duration (in seconds) of the chapter
`position`	Yes	The position of the chapter inside the main content (starting from 1)
`startTime`	Yes	The offset inside the main content where the chapter starts

Update the `getChapterInfo` method to retrieve properties or call the required APIs.

The following is an example of a valid video player plugin delegate:

```
function SampleVideoPlayerPluginDelegate(player) {

    this._player = player;

}


SampleVideoPlayerPluginDelegate.prototype.getVideoInfo = function() {

    var videoInfo = new VideoInfo();

    videoInfo.id = this._player.getVideoId(); // e.g. “vid123-a”

    videoInfo.name = this._player.getVideoName(); // e.g. “My sample video”

    videoInfo.length = this._player.getVideoLength(); // e.g. 240 seconds

    videoInfo.streamType = AssetType.ASSET_TYPE_VOD;

    videoInfo.playerName = this._player.getName(); // e.g. “Sample video player”

    videoInfo.playhead = this._player.getCurrentPlayhead(); // e.g. 115 (obtained from the
video player)
```

```
    return videoInfo;
};


SampleVideoPlayerPluginDelegate.prototype.getAdBreakInfo = function() {
    return null; // no ads in this scenario
};


SampleVideoPlayerPluginDelegate.prototype.getAdInfo = function() {
    return null; // no ads in this scenario
};


SampleVideoPlayerPluginDelegate.prototype.getChapterInfo = function() {
    return null; // no chapters in this scenario
};


SampleVideoPlayerPluginDelegate.prototype.getQoSInfo = function() {
    return null; // no QoS information in this sample
};
```

Next step: ***Configure the VideoHeartbeat library***

Attaching Custom Metadata

The VideoHeartbeat library provides support for custom metadata to be attached to the analytics calls. The relevant APIs for this functionality are defined on the `AdobeAnalyticsPlugin`:

```
AdobeAnalyticsPlugin.prototype.setVideoMetadata = function(data) {};
AdobeAnalyticsPlugin.prototype.setAdMetadata = function(data) {};
AdobeAnalyticsPlugin.prototype.setChapterMetadata = function(data) {};
```

The integration code may call these methods on the `AdobeAnalyticsPlugin` to set custom metadata for the video, the ad and/or the chapter. Note that the metadata for the video will automatically be associated with the ads and chapters as well.

You need to set the metadata prior to calling the relevant `track...()` method on the `VideoPlayerPlugin`, as follows:

- Set the video metadata before calling `trackVideoLoad()`
- Set the ad metadata before calling `trackAdStart()`
- Set the chapter metadata before calling `trackChapterStart()`

This will ensure that the metadata is taken into consideration by the VideoHeartbeat library when processing the `track...()` call.

The code snippet below illustrates how to set custom metadata for video, ads and chapters:

```
// Before calling trackVideoLoad():
adobeAnalyticsPlugin.setVideoMetadata({
    isUserLoggedIn: "false",
```

```
    tvStation: "Sample TV station",
    programmer: "Sample programmer"
});

// [...]

// Before calling trackAdStart():
adobeAnalyticsPlugin.setAdMetadata({
    affiliate: "Sample affiliate",
    campaign: "Sample ad campaign"
});

// [...]

// Before calling trackChapterStart():
adobeAnalyticsPlugin.setChapterMetadata({
    segmentType: "Sample segment type"
});
```

Note: Clearing the custom metadata *- The custom metadata set on the* `AdobeAnalyticsPlugin` *is persistent. It is not reset automatically by the VideoHeartbeat library. To clear the custom metadata, you can pass NULL as the input argument for each of the* `set...Metadata()` *methods. For example, you should do this for ads and chapters once they are complete. Otherwise, the custom metadata will be applied to subsequent ads / chapters. It is your responsibility to ensure that the appropriate metadata is set before the* `trackVideoLoad()` / `trackAdStart()` / `trackChapterStart()` *call.*

Configure the VideoHeartbeat library

After you *Implement VideoPlayerPluginDelegate*, you are ready to add the video heartbeat code to your project. Before you proceed, make sure you have the following:

- A configured `AppMeasurement` object that uses the Marketing Cloud Visitor ID Service.
- An instance of your custom `VideoPlayerPluginDelegate` object.
- Your publisher ID (assigned by Adobe).

On each HTML page where you are tracking video, add a `<script>` tag with a reference to `VideoHeartbeat.min.js`:

```
<script src="VideoHeartbeat.min.js"></script>
```

The following code sample illustrates how to instantiate and configure the video heartbeat components:

```
// Video Player plugin

var vpPluginDelegate = new CustomVideoPlayerPluginDelegate(<my-player>);

var vpPlugin = new VideoPlayerPlugin(vpPluginDelegate);

var vpPluginConfig = new VideoPlayerPluginConfig();

vpPluginConfig.debugLogging = true; // set this to false for production apps.

vpPlugin.configure(vpPluginConfig);


// Adobe Analytics plugin

var aaPluginDelegate = new CustomAdobeAnalyticsPluginDelegate();

var aaPlugin = new AdobeAnalyticsPlugin(appMeasurement, aaPluginDelegate);

var aaPluginConfig = new AdobeAnalyticsPluginConfig();

aaPluginConfig.channel = <syndication-channel>;
```

```
aaPluginConfig.debugLogging = true; // set this to false for production apps.
aaPlugin.configure(aaPluginConfig);

// Adobe Heartbeat plugin
var ahPluginDelegate = new CustomAdobeHeartbeatPluginDelegate();
var ahPlugin = new AdobeAnalyticsPlugin(ahPluginDelegate);
var ahPluginConfig = new AdobeHeartbeatPluginConfig(<tracking-server>, <publisher>);
ahPluginConfig.ovp = <online-video-platform-name>;
ahPluginConfig.sdk = <player-SDK-version>;
ahPluginConfig.debugLogging = true; // set this to false for production apps.
ahPlugin.configure(ahPluginConfig);

// Heartbeat
var plugins = [vpPlugin, aaPlugin, ahPlugin];
var heartbeatDelegate = new CustomHeartbeatDelegate();
var heartbeat = new Heartbeat(heartbeatDelegate, plugins);
var heartbeatConfig = new HeartbeatConfig();
heartbeatConfig.debugLogging = true; // set this to false for production apps.
heartbeat.configure(heartbeatConfig);
```

The configuration of each of the VideoHeartbeat components follows the builder pattern:

- A configuration object is built
- The configuration object is passed as a parameter to the configure method of the component

The list below describes all the configuration parameters:

- **VideoPlayerPlugin**
  - debugLogging: activates logging inside this plugin. Optional. Default value: **false**
- **AdobeAnalyticsPlugin**
  - channel: the name of the syndication channel. Optional. Default value: **the empty string**
  - debugLogging: activates logging inside this plugin. Optional. Default value: **false**
- **AdobeHeartbeatPlugin**
  - **trackingServer:** the server to which all the heartbeat calls are sent. Mandatory. Use the value provided by your Adobe consultant.
  - **publisher:** the name of the publisher. Mandatory. Use the value provided by your Adobe consultant.
  - ssl: Indicates whether the heartbeat calls should be made over HTTPS. Optional. Default value: **false**
  - ovp: the name of the online video platform through which content gets distributed. Optional. Default value: **"unknown"**
  - sdk: the version of the video player app/SDK. Optional. Default value: **"unknown"**
  - quietMode: activates the "quiet" mode of operation, in which all output HTTP calls are suppressed. Default value: **false**

- debugLogging: activates logging inside this plugin. Optional. Default value: false

- **Heartbeat**
  - debugLogging: activates logging within the core Heartbeat component. Optional. Default value: **false**

Note: *Setting the* `debugLogging` *flag to true on any of the VideoHeartbeat components will activate fairly extensive tracing messaging which may impact performance. While these messages are useful during development and debugging, you should set all debugLogging flags to false for the production version of your player app. Note that the* `debugLogging` *flags default to false, so logging is disabled by default.*

Test Your Configuration

Before you continue, load your code in a browser to make sure everything loads without errors. Optionally, set the `debugLogging` flag to `true` while you test:

```
heartbeatConfig.debugLogging = true; // remove or set to false for production!
```

Next, open your code in a browser and check the JavaScript console for errors (the code must be running on a local or remote web server). If there are no errors, you can use the JavaScript console to make a call to `trackVideoLoad()` and then `trackPlay()` to simulate a video play. If you check the network tab, you'll see a call to your Analytics data collection server, and additional calls to `heartbeats.omtrdc.net`.

After you have tested your configuration, continue to *Track Player Events*.

Next step: *Track Player Events*

Track Player Events

Media players that provide JavaScript event handlers are typically tracked by attaching callback functions to the video player event handlers.

The next step is to call the video heartbeat track methods when specific events occur in your player. This typically involves subscribing to events, registering a callback function, and then calling the correct method in the callback. Review the *Track Methods and Player Events* sections for details on exactly which method you should call for each corresponding player event.

The following example demonstrates event handling for HTML 5 video:

```
var myvideo = document.getElementById('movie');

myvideo.addEventListener('play',trackPlay,false);

myvideo.addEventListener('ended',trackComplete,false);

myvideo.addEventListener('seeked', seekEnd, false);

myvideo.addEventListener('seeking', seekStart, false);

myvideo.addEventListener('pause', pause, false);

myvideo.addEventListener('ended', complete, false);


function trackPlay() {
 var myvideo = document.getElementById('movie');
 if (myvideo.currentTime == 0) {
```

```
  vpPlugin.trackVideoLoad();
  vpPlugin.trackPlay();
 } else {
  vpPlugin.trackPlay();
 }
}
function pause(e) {
 vpPlugin.trackPause();
}
function seekStart(e) {
 vpPlugin.trackSeekStart();
}
function seekEnd(e) {
 vpPlugin.trackSeekComplete();
}
function trackComplete() {
 vpPlugin.trackComplete();
 vpPlugin.trackVideoUnload();
}
```

The following example demonstrates event handling for a YouTube player:

```
function onYouTubePlayerReady(id){
    player = document.getElementById("ytplayer");
    if (player.addEventListener) {
        player.addEventListener('onStateChange', 'handlePlayerStateChange');
    }
    else {
        player.attachEvent('onStateChange', 'handlePlayerStateChange');
    }
}


function handlePlayerStateChange (state) {
    switch (state) {
        case 1:
        case 3:
            // Video has begun playing/buffering
```

```
            if (player.getCurrentTime() == 0) {
                vpPlugin.trackVideoLoad();
                vpPlugin.trackPlay();
            } else {
                vpPlugin.trackPlay();
            }
            break;
        case 2:
            vpPlugin.trackPause();
        case 0:
            // Video has been paused/ended
            vpPlugin.trackComplete();
            vpPlugin.trackVideoUnload();
            break;
    }
}
```

Note that each player provides a different way to listen to events. Use the documentation provided by the player API to determine how to listen for player events.

Next step: ***Test Your Video Measurement Code***

Test Your Video Measurement Code

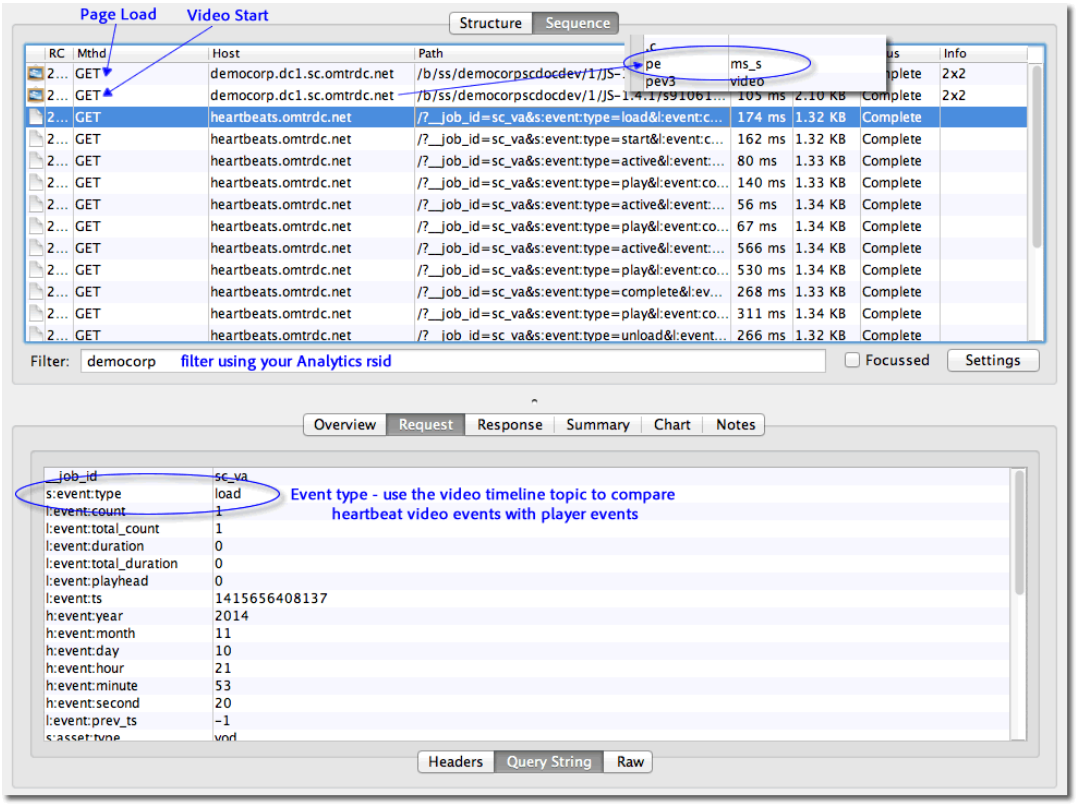
A simple way to test your video heartbeat implementation is to run the code in a demo environment.

1. Load your code in a test environment and use a *packet analyzer* to verify that Analytics server calls and heartbeat calls are being sent. You should see an initial call to your data collection server, and then multiple calls to `http://heartbeats.omtrdc.net`.

   In the initial call to your data collection server:

   - Verify that `pe=ms_s`.

2. Test your implementation throughly to make sure you haven't missed any events. For example, if your player provides a pause event handler and you do not call `trackPause`, your time played metrics will be inflated.
3. In a packet analyzer, inspect the calls and use the *Video Measurement Timeline* to make sure events are being sent as expected. For example, you should see an `s:event:type` of `load` and then `start` when the video begins, and `complete` and then `unload` events when the video completes.



Debug Logging

The VideoHeartbeat library provides an extensive tracing/logging mechanism that is put in place throughout the entire video-tracking stack. You can enable or disable this logging for each VideoHeartbeat component by setting the debugLogging flag on the configuration object.

The log messages follow this format:

```
Format: [<timestamp>] [<level>] [<tag>] [<message>]
Example: [16:01:48 GMT+0200.848] [INFO]
[com.adobe.primetime.va.plugins.videoplayer::VideoPlayerPlugin] \
  Data from delegate > ChapterInfo: name=First chapter, length=15, position=1, startTime=0
```

There are several sections delimited by pairs of square brackets as follows:

- **timestamp:** This is the current CPU time (time-zoned for GMT)
- **level:** There are 4 message levels defined:
  - INFO – Usually the input data from the application (validate player name, video ID, etc.)
  - DEBUG – Debug logs, used by the developers to debug more complex issues
  - WARN – Indicates potential integration/configuration errors or Heartbeats SDK bugs
  - ERROR – Indicates important integration errors or Heartbeats SDK bugs
- **tag:** The name of the sub-component that issued the log message (usually the class name)
- **message:** The actual trace message

You can use the logs output by the VideoHeartbeat library to verify the implementation. A good strategy is to search through the logs for the string `#track`. This will highlight all the `track...()` APIs called by your application.

For instance, this is what the logs filtered for `#track` could look like:

```
[17:47:48 GMT+0200 (EET).942] [INFO] [plugin::player] #trackVideoLoad()
[17:47:48 GMT+0200 (EET).945] [INFO] [plugin::player] #trackPlay()
[17:47:48 GMT+0200 (EET).945] [INFO] [plugin::player] #trackPlay() > Tracking session auto-start.
[17:47:48 GMT+0200 (EET).945] [INFO] [plugin::player] #trackSessionStart()
[17:47:49 GMT+0200 (EET).446] [INFO] [plugin::player] #trackChapterStart()
[17:47:49 GMT+0200 (EET).446] [INFO] [plugin::player] #trackChapterComplete()
[17:48:10 GMT+0200 (EET).771] [INFO] [plugin::player] #trackComplete()
[17:48:10 GMT+0200 (EET).774] [INFO] [plugin::player] #trackVideoUnload()
```

Using this validation method, you can easily spot implementation issues (e.g., the integration code never calls `trackAdComplete()` when an ad completes playback).

Custom Log Writer

If you need to, you can replace the default log message writer (provided by the VideoHeartbeat library) with a custom log message writer. The Logger class has a `setLogWriter()` method which allows for the specification of a custom `ILogWriter`.

Each VideoHeartbeat component uses its own logger object (instance of Logger), so you can change the log writer independently for each component. The `ILogWriter` interface is defined as follows:

```
/**
 * @interface
 */
function ILogWriter() {}

ILogWriter.prototype.write = function(message) {
    throw new Error("Implementation error: Method must be overridden.");
};
```

Transitioning from version 1.4

This section outlines the changes introduced to the VideoHeartbeat library in version 1.5.

Packaging

The previous version (v1.4) of the VideoHeartbeat delivery package contains two separate binary components:

- VideoHeartbeat
- AdobeAnalyticsPlugin

In version 1.5, while these component are still separated at the public API level, they are bundled inside a single library called **VideoHeartbeat**.

VideoHeartbeat components

In version 1.4 there were two components that had to be instantiated and configured:

- **VideoHeartbeat**
- **AdobeAnalyticsPlugin**

In version 1.5, the VideoHeartbeat core has been split into several components:

- **Heartbeat** (the core) - This used to be called VideoHeartbeat in version 1.4

- **AdobeHeartbeatPlugin** - This used to be inside the VideoHeartbeat component. It is responsible for processing the tracking data and sending heartbeat calls.
- **VideoPlayerPlugin** - This used to be inside the VideoHeartbeat component. It is responsible for collecting tracking data from the video player.
- **AdobeAnalyticsPlugin** - This has been a separate plugin since version 1.4. It is responsible for sending calls to SiteCatalyst.

Collecting Video Player Data

In version 1.4, data from the VideoPlayer was gathered via the **`PlayerDelegate`**. You extended the **`PlayerDelegate`** abstract class and provided it as a parameter to the **`VideoHeartbeat`** instance.

The `track...()` methods were exposed by the **`VideoHeartbeat`** class.

With the new component structure, things have changed slightly, as follows:

- The **`PlayerDelegate`** is now called **`VideoPlayerPluginDelegate`**. It must now be provided as a parameter to the constructor method of the VideoPlayer plugin class.
- The **`track...()`** methods are now exposed by the **`VideoPlayerPlugin`**.

New Features and API Changes

This is a list of the new features and APIs that are available in version 1.5:

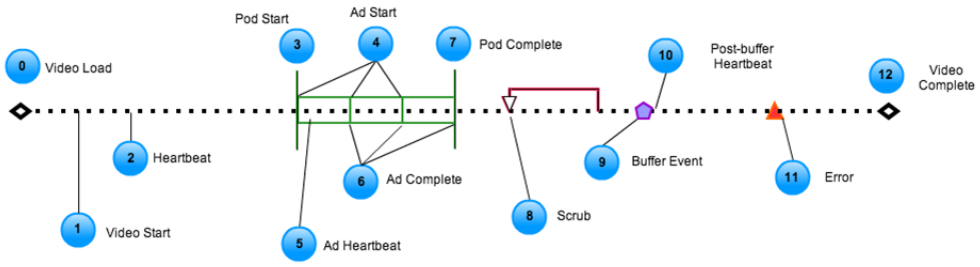
- Support for sending custom metadata
- Support for specifying the **`startupTime`** QoS metric
- New `track...()` method: `trackSessionStart()`
  - This is the method called by the integration code to signal the intention to start playback. It is used to compute the `startupTime` in case it is not provided explicitly on `QoSInfo`.
- The `trackComplete()` method now takes a callback parameter. This callback will be called once the `complete` heartbeat call has been sent over the wire.
- New delegates (one for each of the VideoHeartbeat components). This change arose naturally due to the splitting of the VideoHeartbeat component into multiple sub-components. The new delegates are:
  - **`HeartbeatDelegate`** (for the core component)
  - **`AdobeAnalyticsPluginDelegate`** (for the AdobeAnalyticsPlugin)
  - **`AdobeHeartbeatPluginDelegate`** (for the AdobeHeartbeatPlugin)
- Ability to enable/disable logging per VideoHeartbeat component
- Ability to customize the VideoHeartbeat logger
- The `jobId` heartbeat configuration parameter is no longer required

Below is the list of the APIs that have been removed:

- `onVideoUnloaded()` callback on the `PlayerDelegate`. This method has been removed. We recommend using the callback on the `trackComplete()` method instead.
- `onError()` callback on the `PlayerDelegate`. The `VideoPlayerPlugin` cannot have errors, so this method has been removed. The other VideoHeartbeat components may still have errors. There is an `onError()` callback defined in each of the other VideoHeartbeat components' delegate.

Video Measurement Timeline

This topic provides an overview of when video data is collected.



Note that the heartbeat frequency interval (10 seconds) is the maximum time between heartbeat calls. Within a 10 second interval, you might see mulitple heartbeat calls based on what is occurring in your video. You will see one or more calls to the heartbeat service every 10 seconds, including the following:

- A video playback event, such as play, buffer, and so on

For example, in a 10 second interval, you might see the following video events:

- Play
- Buffer
- Bitrate change
- Ad start
- Ad play

This would indicate that during this interval the user is active, is playing the main asset, encounters a buffering which leads to bitrate change, an ad starts and then plays. You can use a *packet analyzer* to view the `s:event:type` in each heartbeat call to determine the video event that occurred.

Event Number	Event Name	Player to Analytics	Player to Heartbeat Collection	Heartbeat Collection to Analytics
0	Video Load	Single call at beginning of the stream (when the user clicks play)	-	Single call with identifying content info. This includes: • page name (if set) • content type • player name (a.media.playername) • view (a.media.view) • length (a.media.length) • name (a.media.name)
1a	Video Start (on Autoplay, or when	Start call sent		-

	user clicks on play button)			
1b	First frame rendered	First play call sent	Single heartbeat when the 1st frame is rendered, to capture startup time	-
2	Heartbeat	-	Regular heartbeat info	-
3	Pod Start	-	-	-
4	Ad Start	Single call at beginning of ad stream	Ad Start	-
5	Ad Heartbeat	-	Just like regular heartbeat, but with content-type = ad and different video name	-
6	Ad Complete	-	Ad Heartbeat	Single call per ad with all relevant ad info
7	Pod Complete		-	-
8	Scrub	-	Regular heartbeat	-
9	Buffer Event	-	Buffer Heartbeat	-
10	Post-buffer Heartbeat	-	Regular heartbeat enhanced with info about the timing and duration of buffer event	-
11	Error	-	Error call with details about the error (type, impact, etc)	-
12	Video Complete	-	Complete call	(~ 2 minutes after last heartbeat is received) Single call is sent by the Heartbeat Collection server to the Analytics server with the time viewed metrics.

How VideoPlayerPluginDelegate Works

Note: *This video player plugin delegate was previously named* `PlayerDelegate` *in version 1.4.*

If you have reviewed the *Track Methods and Player Events* topic, you might have noticed that none of the `track` methods take any parameters. Instead of passing video name, playhead information, and chapter information directly to these methods, video heartbeat uses a `VideoPlayerPluginDelegate` class that is queried for this information instead. As part of your implementation, you are required to extend this class to provide specific information about your player.

To understand the interaction between the player event listeners, the track functions, and the `VideoPlayerPluginDelegate`, consider the following example.

Event Listeners

When tracking HTML 5 video, you might subscribe to the `play` event using the following code:

```
var myvideo = document.getElementById('movie');
    myvideo.addEventListener('play',trackVideoPlay,false);
```

VideoPlayerPlugin Track Functions

In the `trackVideoPlay()` JavaScript function you assigned to handle the `play` event, you would call `VideoPlayerPlugin.trackPlay()` to let video heartbeat know that playback has started:

```
function trackVideoPlay() {
    VideoPlayerPlugin.trackPlay();
};
```

Note that no video information is passed to the `trackPlay()`.

VideoPlayerPluginDelegate

When the video heartbeat `track...` methods are called, your implementation of `VideoPlayerPluginDelegate` is queried automatically as needed to provide any required details about the video, ad, or chapter. This removes the need for you to determine exactly what information is needed by each track function, you can provide a single object that returns the most current information available. The following is a simple example:

```
function SampleVideoPlayerPluginDelegate(player) {
    this._player = player;
}

SampleVideoPlayerPluginDelegate.prototype.getVideoInfo = function() {
    var videoInfo = new VideoInfo();
    videoInfo.id = this._player.getVideoId(); // e.g. “vid123-a”
    videoInfo.name = this._player.getVideoName(); // e.g. “My sample video”
    videoInfo.length = this._player.getVideoLength(); // e.g. 240 seconds
    videoInfo.streamType = AssetType.ASSET_TYPE_VOD;
    videoInfo.playerName = this._player.getName(); // e.g. “Sample video player”
    videoInfo.playhead = this._player.getCurrentPlayhead(); // e.g. 115 (obtained from the
video player)
    return videoInfo;
};

SampleVideoPlayerPluginDelegate.prototype.getAdBreakInfo = function() {
    return null; // no ads in this scenario
};

SampleVideoPlayerPluginDelegate.prototype.getAdInfo = function() {
    return null; // no ads in this scenario
};

SampleVideoPlayerPluginDelegate.prototype.getChapterInfo = function() {
    return null; // no chapters in this scenario
```

```
};

SampleVideoPlayerPluginDelegate.prototype.getQoSInfo = function() {
    return null; // no QoS information in this sample
};
```

Note: *The `onError()` callback that was part of the `PlayerDelegate` in version 1.4 is removed from the `VideoPlayerPluginDelegate` in version 1.5.*

In this example, when `VideoPlayerPlugin.trackPlay()` is called, your instance of `VideoInfo` is read to determine the current offset of the video to calculate time played. The querying happens automatically, you are required only to extend `VideoPlayerPluginDelegate` and provide an instance of the extended class as a parameter to `VideoPlayerPlugin` when you initialize video heartbeat.

Make sure you take a close look at the sample players to see how `VideoPlayerPluginDelegate` is extended.

Track Methods and Player Events

The video player being instrumented must be capable of triggering a series of events through which any subscriber can be informed about what happens inside the video player. The following tables present the one-to-one correspondence between player events and the associated call exposed by the public API of the video heartbeats library.

Video Playback

Event	Method Call	Parameter List
Load the main video asset	`trackVideoLoad()`	None
Unload the main video asset	`trackVideoUnload()`	None
Autoplay ON, or user clicks play	`trackSessionStart()`	None
Playback start	`trackPlay()`	None
Playback stop/pause	`trackPause()`	None
Playback complete	`trackComplete()`	None
Seek start	`trackSeekStart()`	None
Seek complete	`trackSeekComplete()`	None
Buffer start	`trackBufferStart()`	None
Buffer complete	`trackBufferComplete()`	None

Rules and Practices

- **Methods to be called in pairs:**

  The following methods must be called in pairs (that is, each `track...Start()` must have a corresponding `track...Complete()`):

  - `trackBufferStart()` and `trackBufferComplete()`
  - `trackPause()` and `trackPlay()` (note that if the player is closed before the pause resumes, the corresponding method might not be called)
  - `trackSeekStart()` and `trackSeekComplete()` (with an exception: there may be multiple `trackSeekStart()` calls before a `trackSeekComplete()`)
  - `trackAdStart()` and `trackAdComplete()` (unless the user seeks out of the ad without playing it to completion)
  - `trackChapterStart()` and `trackChapterComplete()` (unless the user seeks out of the chapter without playing it to completion)

  The `track...Start()` call is not required to be followed by a `track...Complete()` call, as there may be other `track...()` method calls in between. For example, the following sequence of `track...()` method calls is valid and describes a user who is seeking through the stream while paused, and resumes playback after two seeks:

```
trackPause(); // Signals that the user paused the playback.
trackSeekStart(); // Signals that the user started a seek operation.
trackSeekStart(); // Signals that the user started another seek operation (before the first
one was completed).
trackSeekComplete(); // Signals that the second seek operation has completed.
trackPlay(); // Signals that the user resumed playback.
```

- **Tracking the completion of content:**

The `trackComplete()` method is used to signal the completion of the video (i.e., the content was played to the end). You should call `trackComplete()` before calling `trackVideoUnload()` if the video was completed. When the user quits the video before its completion (e.g., by switching to another video in a playlist), you should not call `trackComplete()`. Instead, you should simply close the tracking session by calling `trackVideoUnload()`.

Ad Playback

Event	Method Call	Parameter List
An ad starts	`trackAdStart()`	None
An ad completes	`trackAdComplete()`	None

The `trackAdStart()` and `trackAdComplete()` methods are the only track methods required in order to signal the beginning and completion of an ad.

You do not need to (and should not) call any additional track methods to signal the transition from ad to content or vice-versa. For instance, you should not signal the pause of the main video (via `trackPause()`) when an ad starts. This is handled automatically by the `VideoPlayerPlugin` when you call `trackAdStart()`

Chapter Tracking

Event	Method Call	Parameter List
A new chapter starts	`trackChapterStart()`	None
A chapter completes	`trackChapterComplete()`	None

QoS Tracking

Event	Method Call	Parameter List
A switch to another bitrate occurs	`trackBitrateChange()`	None

Error Tracking

Event	Method Call	Parameter List
An error occurs at the player level	`trackVideoPlayerError()`	`errorId:String` - unique error identifier
An error occurs at the application level	`trackApplicationError()`	`errorId:String`

Copyright

Adobe Primetime Video Heartbeat SDK Guide

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.