Ad Management SDK for JavaScript

User Guide

RELEASE: v1.7.0

Copyright © 2009-2018 Yospace Technologies Ltd.

All rights reserved. The information contained within this document is confidential and forms a proprietary trade secret of Yospace Technologies Ltd.

The software described within the document is supplied under a licence agreement and may only be used in accordance with the terms and conditions set out within.

Disclosure of this document, or of any information contained therein, to parties without written permission from Yospace, is expressly forbidden. For permission requests, write to **Yospace Legal**.

This document is subject to alteration without notice and does not represent a commitment on the part of Yospace Technologies Ltd.

Table of Contents

1. **Introduction** 5
   1.1. **How to Read this Document** 5
2. **Getting Started** 6
3. **Supported Playback Modes** 7
   3.1. **Live Playback** 7
   3.2. **Live Pause Playback** 7
   3.3. **VOD Playback** 8
   3.4. **Non-Linear Start Over (NLSO) Playback** 8
4. **Overview of the Ad Management SDK** 9
5. **Configuring Playback Policy** 10
6. **Initialising the SDK** 11
   6.1. **Setting the Configuration Parameters** 12
   6.2. **Code Fragment invoking a Factory Method** 13
   6.3. **Factory Method Initialisation** 14
   6.4. **Session Status Notification** 15
   6.5. **Playing the Stream** 15
   6.6. **Initialisation Errors** 15
   6.7. **Creating an AJAX Delegate** 16
      6.7.1. **JavaScript XML Abstraction** 17
         6.7.1.1. **XML Document Parsing** 18
7. **Tracking Playback in the SDK** 21
   7.1. **Reporting Player States** 21
      7.1.1. **Handling Clickable Elements** 23
         7.1.1.1. **Clickable Linear Creatives** 24
         7.1.1.2. **Clickable Non-Linear Creatives** 24
      7.1.2. **Pause and Resume for Live Pause** 25
   7.2. **Interpreting and Delivering Metadata** 26
      7.2.1. **ID3 Timed Metadata for HLS and MPEG-TS Streams** 26
      7.2.2. **EMSG in MPEG-DASH** 27
      7.2.3. **Delivery of Metadata to the SDK** 28
         7.2.3.1. **Important Note about Presentation Time Stamps** 28
   7.3. **Supported VAST Tracking Beacons** 29
8. **Handling Analytic Callbacks from the SDK** 31
   8.1. **Content and Ad Transition Callbacks** 31
   8.2. **Timeline Information Callbacks** 32
      8.2.1. **Working with the Timeline** 32
         8.2.1.1. **Implementing UI Scrub Control** 33
         8.2.1.2. **Updating the Timeline** 33
            8.2.1.2.1. **Constructing a Dynamic Timeline for Live Pause** 34
   8.3. **Inspecting the Break currently playing** 35
   8.4. **Inspecting the Ad currently playing** 36
   8.5. **Other Callbacks** 36
9. **Playback Policy** 38
   9.1. **Policy for Linear Playback** 39

 Copyright © 2009-2018 Yospace Technologies Ltd.

9.2. **Policy for Non-Linear Playback** 39
9.3. **Skip Offset** 39
10. **Suppressing Analytics** 41
11. **Implementing VPAID in Live and VOD Playback** 42
11.1. **VPAID Implementation Procedure** 43
11.1.1. **Implementing the IAB VPAID Interface for JavaScript** 44
11.1.1.1. **Methods** 44
11.1.1.2. **Properties** 45
11.1.1.3. **Events** 46
12. **Viewability Adapter** 48
12.1. **Viewability Preparation and Integration** 48
13. **VAST Macros** 49
14. **Cleaning Up** 50
15. **Running in Debug Mode** 51
16. **Validating your App prior to Release** 52
17. **Language Reference** 53
18. **Glossary** 54

 Copyright © 2009-2018 Yospace Technologies Ltd.

1. Introduction

The Yospace Ad Management SDK is a platform library that is embedded in a customer's application to provide analytics and playback policy management for streams that utilise Yospace's Server-Side Ad Insertion (SSAI).

The SDK supports the use of clickable linear content and dynamic overlays in both ***Video On Demand*** (VOD) and Live streaming. It also facilitates the implementation of playback policy to provide UI ***Scrub Control*** and protect advert playback for ***Non-Linear*** streams.

For any **Supported Playback Modes [7]** that involve tracking the playhead position - that is to say, **VOD Playback [8]**, **Non-Linear Start Over (NLSO) Playback [8]** and **Live Pause Playback [7]** - use of the SDK is mandatory, in order to fire tracking events and implement playback policy.

1.1. How to Read this Document

You should read the **Getting Started [6]** chapter before reading any other parts of the document. This makes reference to the sample application(s), provided with relevant playback and policy defaults that will help you set up and use the SDK in your own application.

Although the remainder of the guide is designed to cover the integration of the SDK in the order in which it might be implemented and configured, it is intended to be used as more of a reference document for developers. The way in which you navigate it will depend on your level of knowledge of the particular operating platform, and of ***yospaceCDS***.

The following common syntactic and typographic conventions are used in this document:

- An 'application' is hereafter referred to as an 'app'.
- An 'advertisement' or 'advert' is hereafter referred to as an 'ad'.
- A monospaced font `like this` is used to indicate class names and platform code.
- Italicised hyperlinks, such as ***yospaceCDS***, provide links to the associated glossary definitions.
- A bold hyperlink takes you either to a class that is fully described in the **Yospace Developer Portal**, or to another section of this document.

 Copyright © 2009-2018 Yospace Technologies Ltd.

2. Getting Started

This section outlines Yospace's recommended high-level approach to using the SDK.

1. Download and run the required Yospace sample app from those provided on the **Yospace Developer Portal**, without making any modifications. This will allow you to verify that the SDK is functioning correctly in a Live stream, and handling analytic events as expected.

   The sample app provided by the JavaScript SDK implements VPAID (refer to **Implementing VPAID in Live and VOD Playback [42]**), Moat/Viewability (refer to **Viewability Adapter [48]**), and a basic 'right to watch' Policy that prohibits seeking during ad breaks in a Non-Linear stream (refer to **Configuring Playback Policy [10]**). The Policy is implemented in its own separate JavaScript file, in unminified form, and referenced by the sample app. You can use this as a model to set up your company's own Playback Policy,

2. Use the documentation and source from the sample app to give you an overall sense of how you might integrate the SDK into your app.

   For the links to the class documentation, refer to **Language Reference [53]**.

3. Verify that the test stream plays back in your existing app, by hard coding the following Yospace test URL into your app: `http://csm-e.cds1.yospace.com/csm/extlive/yospace02,hlssample.m3u8?yo.ac=true`.
4. If the test stream results in an error, then consult the troubleshooting in the sample app documentation and repeat the above steps. If the error persists, contact **support@yospace.com**.
5. If/when there are no errors, integrate the SDK into your app.

   If your platform does not have a full Asynchronous JavaScript and XML (AJAX) implementation, then you will need to create an AJAX container (refer to **Creating an AJAX Delegate [16]**).

6. Play back your own stream in your app, and wait until the first ad break in order to verify that the required analytics are being generated. If you are running in VOD playback mode, also verify that you can view the historical ad breaks in your app. If the output is not as expected, then contact **support@yospace.com**.

The sample app is only a suggested starting point for the integration, and you should always consult the instructions in the remainder of this guide.

 Copyright © 2009-2018 Yospace Technologies Ltd.

3. Supported Playback Modes

The SDK library supports Live Playback, *Live Pause*, ***Video On Demand*** and Non-Linear Start Over (***NLSO***).

3.1. Live Playback

Prior to an ad break, the library receives ***Video Ad Serving Template*** (VAST) or ***Video Ad Serving Template*** (VMAP) XML data from the Yospace ***Central Streaming Manager*** (CSM).

During ad sections of the video stream, the library delivers:

- information to the app about:
  - ad breaks;
  - ad begin and end points;
  - clickable ads (linear and overlays), including 'click-thru' URLs.
- analytic notifications to a remote source, for example timed events such as:
  - quartiles;
  - impressions.

3.2. Live Pause Playback

The Yospace platform allows ***Live Pause*** playback, whereby a Live stream can be paused for up to 90 minutes. On resumption, the stream plays back from the point at which it was paused, provided that this remains within the 90-minute window. Because the DVR window is only configurable for up to 90 minutes, this means that, if the stream is paused for longer than this, the behaviour **must be** defined by the app: for example, the app could cause the stream to resume automatically when the 90-minute window is reached, or it could cause the playhead to skip forward.

The Ad Management SDK supports the app in protecting against skipping over any ad breaks that may have aired while the stream was paused. Refer to **Configuring Playback Policy [10]** and **Playback Policy [38]**.

In terms of support for client apps, the difference between Live Playback and Live Pause is that, for Live Pause, your app is informed about video stream timeline updates, with a start time and end time representing the pause window and any ad breaks within that window.

In Live Pause mode, ID3 metadata tags (refer to **Interpreting and Delivering Metadata [26]**) are not used, even when playing at the Live point. The SDK is instead driven by playhead position, as with a ***Non-Linear*** stream.

Copyright © 2009-2018 Yospace Technologies Ltd.

3.3. VOD Playback

As part of the library initialisation phase, the SDK receives ***Video Multiple Ad Playlist*** (VMAP) XML data from the Yospace Session Access Service, with proprietary extensions, representing the entire stream.

During the ad sections of the video stream, the library delivers the same notifications and information to remote and local sources, the exact composition of which depends on the delivered VMAP/VAST.

3.4. Non-Linear Start Over (NLSO) Playback

NLSO is an enhanced version of the time-shifted viewing of live streams via the 'start over' feature, which allows the viewer to play from the beginning of a programme in a 'linear manner'. NLSO allows the viewer to use ***Trick Play*** while playing programme parts that have been previously aired.

In standard 'start over' streaming, because the playback experience is strictly Linear, ads are replaced using the same policy as 'live' replacement: that is to say, the break is replaced by content that matches the duration of the underlying broadcast break. In NLSO, the replaced breaks can be expanded or contracted in duration, depending on which ads have been returned by the ad server.

The SDK can be used to determine the viewer's ability to navigate within the stream, typically blocking the ability to use ***Scrub Control*** to move the playhead past an ad break that has not yet been viewed (refer to **Policy for Non-Linear Playback [39]**). This is very much as one might expect standard VOD mid-roll ad insertion to work, the difference being that the SDK regularly polls the ***Central Streaming Manager*** (CSM) for additional data as the NLSO playlist grows.

 Copyright © 2009-2018 Yospace Technologies Ltd.

4. Overview of the Ad Management SDK

The Yospace Ad Management SDK allows you to set up a session with the Yospace ***Central Streaming Manager*** (CSM) so that the app can deliver the resultant SSAI URL to the player, and the SDK can track playback in order to fire the required beacons.

The process for Live playback can be illustrated as follows:

Ad Server
Yospace CSM
Yospace Ad Management SDK
Player
Request Master Manifest
Initialise SDK with Playback URL
Send Master Manifest
Loop
Ad Server Call
Ads
VAST Response
Content
Ads
Extract Data
Session ID
Analysis URL
Live Pause URL
Start Playback
User Interaction
Notify Play/Pause/Resume/FullScreen, etc.
Provide regular playhead position updates
Metadata (ID3/EMSG)
Forward Metadata
Ads
Fire Tracking Pixels
Ad break Started
Background Task Loop
Poll for upcoming VAST data
Poll VAST response for upcoming Ads
Ads are built into internal ad bucket

The Ad Management SDK can introspect analytics data both prior to, and during, playback, and also send analytic reports to a remote source at certain points during playback. These reports may include data such as the percentage of video content watched, the number of ads watched, and so on. The library can also provide this data to multiple interested observers, including the app in which it is embedded, via a broadcast, delegate or callback mechanism.

The SDK for JavaScript is supplied as a minified `.js` file to be deployed together with a customer application. Interaction from the customer application is typically made via a series of simple method calls on a handful of available class types.

Reporting back to the app is handled via the **registerPlayer** method, which is used to register a player object that contains a set of standard callback functions, invoked at the appropriate points in time. This is covered as part of the session initialisation in the examples in **Code Fragment invoking a Factory Method [13]**.

Copyright © 2009-2018 Yospace Technologies Ltd.

5. Configuring Playback Policy

Playback policy is key to enabling the control of playback and UI behaviour in the client app. For example, an app may be set up to enquire as to whether pausing in an ad is permitted, or whether the use of ***Seek to*** is permitted during ad breaks in a Non-Linear stream.

Before **Initialising the SDK [11]**, you must ensure that you have configured your own organisation's playback policy, particularly in the case of Non-Linear streams (refer to **Policy for Non-Linear Playback [39]**). You are advised to take the sample app that is supplied as a basis (refer to **Getting Started [6]**) and then modify it as required to suit your organisation's particular business requirements.

The sample policy implementation is implemented via the **YSPlayerPolicy** class, which is not included in the minified SDK code, but is provided separately, in unminified form. This will implement the default behaviour detailed below and will be instantiated automatically when a session is started.

The sample implementation has the following defaults:

- For Live playback, `canSkip`, `canSeek` and `canPause` are all set to `FALSE` by default. It is recommended that you leave them this way, in accordance with **Policy for Linear Playback [39]**.
- For Non-Linear streams, the sample policy implements 'right to watch', which means that a viewer needs to watch any ad break directly prior to the content that they wish to view, in order for playback of that content to be allowed.

The policy implementation must always be included as part of the deployment of any app that uses the SDK - otherwise, the SDK will not function.

Please refer to **Playback Policy [38]** for further details.

 Copyright © 2009-2018 Yospace Technologies Ltd.

6. Initialising the SDK

Initialisation should be carried out by calling the factory method for the required stream. There is a factory method available for each of the **Supported Playback Modes [7]**:

Factory Method	Description
`createForLive`	Implements a Live stream that is typically delivered in a 'moving window'-style playlist (either M3U8 or MPD). The player polls the playlist regularly to observe new segments being added to the end of the playlist, and expired segments being removed from the head of the playlist. Pausing and scrubbing are not supported. Analytics information is fetched periodically by the SDK from the Yospace CSM. Because there is no positional frame-of-reference, the SDK requires that the client application reports all in-band metadata contained within the stream (from either ID3 or EMSG packets) at the correct presentation time contained within the metadata packet.
`createForLivePause`	Similar to `createForLive`, this also implements a live stream delivered in a 'moving window'-style playlist (either M3U8 or MPD). The player polls the playlist regularly to observe new segments being added to, and expired segments being removed from, the playlist at each poll. The key difference when compared with 'standard' live playback is that a longer DVR window is made available, which effectively makes it possible for the client to pause playback of the Live stream, and to scrub within the DVR window. Analytics information is fetched periodically by the SDK from the Yospace CSM server in VMAP format. Because there is positional information, there is no need to report metadata to the SDK as it will rely on the playhead position information, which the client application must report very regularly to the SDK (usually at intervals of no greater than 500 milliseconds). A timeline representation is made available to the client application so that it can understand the nature of the content that is available in the current DVR window.

Copyright © 2009-2018 Yospace Technologies Ltd.

Factory Method	Description
`createForNonLinear`	Implements a **Non-Linear Start-Over** stream (refer to **Non-Linear Start Over (NLSO) Playback [8]**), which is a hybrid of a **Live/VOD** stream. At stream start, the entire program up to the current live point is reported to the application, and the existing breaks from this **VOD** portion are resolved (typically using shorter break durations). The viewer watches the stream from the start of the programme and the player continues to poll for updated M3U8/MPD manifests to receive new fragments being added at the live point of the stream. Because the resolved breaks are shorter in the **VOD** portion, the viewer will eventually 'catch up' with the live point of the stream, at which point they will continue watching the live part of the stream in real-time. Pausing and scrubbing are permitted (although the client may choose to implement a 'right to watch' policy in accordance with their own business needs, which, for example, prohibits scrubbing through ads). Analytics information is fetched periodically by the SDK from the Yospace CSM in VMAP format. Because there is positional information, there is no need to report metadata to the SDK, as it will rely on the playhead position information which the client application must report very regularly to the SDK (usually at intervals of no greater than 500 milliseconds). A timeline representation is made available to the client application so that it can understand the nature of the content that is available in the current DVR window.
`createForVOD`	Implements a **VOD** stream, which is a Linear piece of content with fixed, known start and end positions. Playback may be paused and scrubbed at any point, and there is no need to poll playlists as the initial fetch is complete and they will not be updated. The SDK will obtain the timeline representation and analytics information at startup in VMAP format, and make a timeline representation available to the application to allow it to see the nature of the content that it is playing. There is no need to deliver metadata to the SDK in this mode, as it relies on the playhead position information being reported regularly to the SDK (typically, at an interval of no greater than 500 milliseconds).

Each method takes a `YSSessionProperties` object as a parameter, where you can set individual properties to modify initialisation flow and behaviour, to ensure that the internal behaviour of the SDK matches the required form of playback.

If you wish the CSM/analytics to return VMAP documents (with ad break tracking events) instead of VAST documents, then set the parameter `yo.av` to `2` in the CSM playback URL. In this mode, events are raised as follows: for Non-Linear playback, they are raised at the point at which the ad break would have been inserted; for Linear playback, they are raised as soon as the SDK has finished parsing the VMAP.

6.1. Setting the Configuration Parameters

The most common configuration parameters are outlined in the following table. Whilst there are other parameters available, they are not documented here as they are intended only for use in exceptional cases.

Copyright © 2009-2018 Yospace Technologies Ltd.

Accessor Name	Type	Description
LOW_FREQ	number	Sets the poll interval (in milliseconds) for analytics calls. This should usually be set to the same value as the target segment duration for the stream being played.
DEBUGGING	bool	Controls whether or not debugging output should be produced. Refer to **Running in Debug Mode [51]**.

The configuration parameters can also be changed globally (across all instances of the `YSSessionManager`) by setting them manually against the `YSSessionManager.DEFAULTS` object, for example:

```
YSSessionManager.DEFAULTS.DEBUGGING = true;
```

6.2. Code Fragment invoking a Factory Method

The following code fragment shows how a factory method may be invoked to start a Live playback session, assuming that configuration parameters have been set globally:

```
// Create a session for playing back a live stream
var sessMgr = YSSessionManager.createForLive
    ("http://csm-e.cds.yospace.com/stream.m3u8", null,
    function(state, result) {
        console.log("SessionManager is Ready. State: " + state);
        // state is a YSSessionResult value
        if (state !== YSSessionResult.INITIALISED) {
            console.log("Response code: " + result);
        } else {
            sessMgr.registerPlayer(cb_obj);
        }
    }
);

var cb_obj = {
   AdBreakStart: function() {
       // Function gets called whenever an advert break starts.
       // This fictitious example causes the video player to display
       // a clickable element
       myPlayer.showClickableElement();
       myPlayer.setOnClick(function() {
           sessMgr.reportPlayerEvent(YSPlayerEvents.CLICK);
       });
   },
   AdvertStart: function(mediaId) {
       // Function gets called at the start of each advert within a break
       // (except for filler content - e.g. ident etc.)
       var advert = sessMgr.session.currentAdvert;
       var ad = advert.advert;   // Obtain the actual underlying ad
       var nonLinears = ad.getNonLinears();
       if ((nonLinears !== null) && (nonLinears.length > 0)) {
           var idx = 0; // This will hold the non-Linear Index
           for (var i = 0; i < nonLinears.length; i++) {
               var item = nonLinears[i];

        // This item now needs to be added as a non-linear to the display.
        var imgs = item.getAllResources()['images'];
        // imgs will now contain an object whose properties are
        // the mime type of available images and the values of the
       // image URLs.
       for (var j in imgs) {
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
        if (imgs.hasOwnProperty(j)) {
        // or hasOwnProperty('image/png') to restrict type
        myPlayer.addNonLinear(imgs[j], idx++, function(_index) {
        sessMgr.reportPlayerEvent(YSPlayerEvents.NONLINEAR, _index);
                        });
                    }
                }
            }
        }
    },
    AdvertEnd: function(mediaId) {
        // Function gets called at the end of each ad within a break
        // (except for filler content - e.g. ident, etc.)
        myPlayer.removeAllNonLinears();
    },
    AdBreakEnd: function() {
        // Function gets called at the end of an ad break, signalling a
        // return to the main content.
        // This fictitious example causes the video player to remove
        // the clickable element that it added previously.
        myPlayer.removeClickableElement();
        myPlayer.setOnClick(null);
    },
    UpdateTimeline: function(timeline) {
        // Function gets called whenever there is updated information
        // concerning the playback timeline (for VOD and Non-Linear
        // content only)
    },
    AnalyticsFired: function(call_id, call_data) {
        // Function gets called whenever an analytics call is made by the SDK.
    }
};
sessMgr.registerPlayer(cb_obj);
```

6.3. Factory Method Initialisation

The factory method, invoked via **Code Fragment invoking a Factory Method [13]**, performs the following steps:

1. Constructs an instance of the Session Manager class.
2. Constructs an appropriate `Session` descendant instance.
3. Binds the Session instance to the Session Manager instance.
4. Returns the reference to the newly created Session Manager instance.

The Session Manager instance cannot actually be used until the remaining two asynchronous steps have completed.

5. Initialises the session and obtain a playback URL.
6. Invokes the callback function, so that the client is aware that, in the event of a zero or negative `result` value (refer to **Session Status Notification [15]**), it can now start to use the Session Manager instance that was handed back to it.

In the case of all methods except `createForLive()`, the playhead position is reported and timeline information is made available. In the case of `createForLive()` , metadata must only be reported when the stream is actively playing. It is therefore essential that the metadata is reported when the playhead reaches the appropriate presentation time, and **not** when the metadata is decoded from the fragment.

 Copyright © 2009-2018 Yospace Technologies Ltd.

6.4. Session Status Notification

Once an attempt has been made to create the **YSSessionManager** instance, the following callback is used:

```
function (state, result)
```

It takes the following parameters:

Parameter	Description
state (YSSessionResult)	Indicates whether or not communication could be established with the CSM, and can be any of the following: • INITIALISED - A session has been successfully established with the CSM. Proceed from **Playing the Stream [15]**. • NOT_INITIALISED - A session with the CSM could not be established owing to an HTTP or network error code. • NO_ANALYTICS - A session with the CSM could not be established owing to a Yospace error code.
state (YSSessionStatus)	The result provides additional information about a result of either NOT_INITIALISED or NO_ANALYTICS. Refer to **Initialisation Errors [15]** for details. In the case of INITIALISED, the result is either zero (0) (meaning no errors), or a negative value denoting NO_LIVEPAUSE (meaning that a connection has been established with the CSM, but that the stream cannot be paused, and so is only valid for Linear playback). You can proceed from **Playing the Stream [15]**.

6.5. Playing the Stream

In the event of a successful initialisation, the client app can retrieve the playback URL from the `Session Manager` instance, in order to start playing the stream with its own media player:

```
var targetUrl = sessMgr.masterPlaylist();
```

It is critical that the target URL is the one used for playback as it will now contain a session identifier that can be matched against the analytics responses from the CSM server.

If you wish the CSM/analytics to return VMAP documents (with ad break tracking events) instead of VAST documents, then set the parameter `yo.av` to `2` in the CSM playback URL. In this mode, events are raised as follows: for Non-Linear playback, they are raised at the point at which the ad break would have been inserted; for Linear playback, they are raised as soon as the SDK has finished parsing the VMAP.

Once playback has started, the client app must post informational feedback to the SDK to ensure that it is kept aware of the state of the stream playback (refer to **Tracking Playback in the SDK [21]**).

6.6. Initialisation Errors

If communication cannot be established with the CSM during initialisation, then, in accordance with the **Session Status Notification [15]**, the SDK will return one of the following:

Copyright © 2009-2018 Yospace Technologies Ltd.

- `NO_ANALYTICS`. Typically caused by the `result` event `NON_YOSPACE_URL`, which means that the primary URL is not a valid Yospace CSM URL, or does not contain the parameter `yo.ac=true`.
- `NOT_INITIALISED`. Typically caused by a network error, indicated via any of the following `result` events:
  - `CONNECTION_ERROR`. A connection could not be established with the CSM.
  - `CONNECTION_TIMEOUT`. Initialisation timed out before a connection could be established with the CSM.
  - `MALFORMED_URL`. The player URL is not correctly formatted.
  - `NON_YOSPACE_URL`. The primary URL is not a valid SSAI stream.
  - A positive integer denoting the `HTTP` status code, for example `404`.

For the next step, refer to **Cleaning Up [50]**.

6.7. Creating an AJAX Delegate

If your platform does not have a full Asynchronous JavaScript and XML (AJAX) implementation, then you will need to create an AJAX container.

The following is commented code outlining a delegate that will result in the `XHR` (`XMLHttpRequest`) object required to execute AJAX in order to play the stream:

```
/**
  * <p>Configuration allows for externalising AJAX calls.<p>
  * <p>When specified, this will be a function used to make AJAX calls.
  * The function prototype is:<p>
  *
  * function(url, props)
  *
  * where url is the URL to be loaded (via http/s get method)
  * and props is an object containing 2 significant properties (all others
  * should be just ignored):
  *
  * onSuccess: function(response)
  * onFailure: function(response)
  *
  * The response parameter supplied should be an object, and
  * must have the following method implemented:
  *
  * getResponseHeader: function(key)
  *
  * This method will return the value of the HTTP header value contained in
  * the response whose key matches that provided.
  * e.g. getResponseHeader("X-YOSPACE-CSM-SESSION-ID") would return the
  * header value for X-YOSPACE-CSM-SESSION-ID.
  *
  * Additionally, the response object must contain the following properties:
  *
  * responseXML      // An XMLDocument instance populated with the response, IF the
  *                  // response is XML
  * responseText     // A string containing a string representation of the response,
  *                  // regardless of whether or not the response is XML
  * transport        // An object containing responseURL and status properties. This property should
  *                     be populated with the actual target URL loaded (this will be the original
  *                     requested URL in most cases, but as result of a 3xx status code, should be
  *                     the redirection target). The status is the HTTP status code.
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
 *
 * @example
 * var myCustomResponse = {
 *      __RESPONSE_HEADERS: {}, // These are just used internally in this example
 *
 *      getResponseHeader: function(key) {
 *          if (this.__RESPONSE_HEADERS.hasOwnProperty(key.toLowerCase())) {
 *              return this.__RESPONSE_HEADERS[key.toLowerCase()];
 *          } else {
 *              return null;
 *          }
 *      },
 *
 *      responseXML : null,
 *
 *      responseText : null,
 *
 *      transport: {
 *          responseURL : ""
 *      }
 * };
 *
 * // An example using jQuery
 * function myCustomAjax(url, props) {
 *      myCustomResponse.transport.responseURL = url;
 *      myCustomResponse.transport.status = 200;
 *      myCustomResponse.responseXML = null;
 *      myCustomResponse.responseText = null;
 *      myCustomResponse.__RESPONSE_HEADERS = {};
 *
 *      // Make call
 *      $.ajax(url, {
 *          success: function(data, textStatus, jqXHR) {
 *              try {
 *                  myCustomResponse.responseXML = $.parseXML(jqXHR.responseText);
 *              } catch(err) {}
 *              myCustomResponse.responseText = jqXHR.responseText;
 *              var headers = jqXHR.getAllResponseHeaders().split("\r\n");
 *              for (var i = 0; i < headers.length; i++) {
 *                  var hdr = headers[i].split(":");
 *                  myCustomResponse.__RESPONSE_HEADERS[hdr[0]] = String(hdr[1]).replace(/\s+/g, "");
 *              }
 *
 *              props.onSuccess(myCustomResponse);
 *          },
 *          error: function(jqXHR, textStatus, errorThrown) {
 *              myCustomResponse.responseText = textStatus;
 *              myCustomResponse.transport.status = jqXHR.status;
 *              props.onFailure(myCustomResponse);
 *          }
 *      })
 * };
```

If you also do not have the requisite XML Document Object Model (DOM), refer to **JavaScript XML Abstraction [17]**.

6.7.1. JavaScript XML Abstraction

The SDK requires that a loader is provided which can be instantiated using the following construct:

```
new ProtoAjax.Request(
        server_url,
        {
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
            method: "get",
            evalJSON: false,
            evalJS: false,
            onSuccess: function(response),
            onFailure: function(errorMsg)
        }
    );
```

where `response` is a response object that can deliver the response either as text or XML (via two properties), and has the ability to read response headers:

```
response.responseText    // Returns a String
    response.responseXML    // Returns an XML Document Object
    response.getResponseHeader(header_key)    // Obtain specific HTTP header value
```

Only the `"get"` method is currently required. `evalJSON` and `evalJS` will always be set to `false` and need not be handled. If `getResponseHeader` cannot be implemented, please contact **support@yospace.com**.

Typically, the `responseXML` object will return an XML Document Object Model (DOM) instance that can be used by the SDK. To help you achieve compatibility with the SDK with minimal changes, the following section (**XML Document Parsing [18]**) describes the functionality expected by the DOM implementation, enabling a wrapper to be presented that encapsulates the XML functionality of the target platform.

It is not vital for the functionality of the SDK that a bona fide XML DOM instance is returned by `responseXML`, just so long as the XML **Document** returned by the loader supports the subset of the DOM specification outlined in **XML Document Parsing [18]**.

6.7.1.1. XML Document Parsing

The DOM classes and associated API elements required to parse the XML **Document** returned by the loader are shown below.

Class	Property/Method	Description/Notes
Document		The **Document** class encapsulates the top-level of the XML interpretation. It must have its `firstChild` as its root node (and only node).
	Properties	
	`nodeType` [number]	This value for a **Document** object is always 9.
	`firstChild` [node]	Returns the first child of the **Document** as type `node`.
	Methods	
	`getElementsByTagName(tag)` `getElementsByTagName(namespace, tag)`	Returns a `NodeList` (or **HTMLCollection [19]**) of all element nodes for which the `TagName` matches the `tag` parameter provided, and for which the `namespace` matches the `namespace` parameter provided. If the `namespace` parameter is not provided, then the `namespace` syntax is ignored and need not be specified as part of the supplied `tag` parameter. If there are no matches, an empty `NodeList` is returned.

 Copyright © 2009-2018 Yospace Technologies Ltd.

Class	Property/Method	Description/Notes
HTMLCollection		The `HTMLCollection` type can be thought of as a simple array of nodes. The number of nodes contained within the array can be obtained via the `number` argument, and a specific node can be retrieved via an `index` number.
	Properties	
	`length` [number]	Returns the number of nodes contained within the current `HTMLCollection`.
	Methods	
	`item(index)` [number]	Returns the **Node [19]** at the `index` number specified via the `index` parameter. All nodes are expected to be indexed sequentially based on their occurrence within the source XML.
Node		
	Properties	
	`nodeType` [number]	The possible meaningful `nodeTypes` are as follows: • 1 - `element node` • 2 - `attribute node` • 3 - `text node` • 4 - `CDATA` section • 8 - `comment` In terms of the SDK, there is really only a need to make a distinction between `text nodes` (value 3) and the other `nodeTypes`.
	`tagName` [string]	The `tag` name of the current node, including any `namespace` (where applicable).
	`localName` [string]	The `tag` name of the current node, excluding any `namespace` (where present).
	`parentNode` [node]	Returns the node of which the current node is a child.
	`firstElementChild` [node]	Returns the first child node that is a node of type `element`. Nodes of type `text` and `comment` are ignored.
	`firstChild` [node]	Returns the first child node. This may be a node of type `element`, `text` or `comment`, depending on which one is first.
	`nextSibling` [node]	Returns the node that appears immediately following the current one, at the same hierarchical level within the same XML block. For example, where a parent node contains two child nodes, the first child will have a `nextSibling`, but the second will not.
	`textContent` [string]	Returns the textual content of the current node (and its descendants).
	`innerHTML` [string]	Returns the textual content of the current element (and its descendants).

Copyright © 2009-2018 Yospace Technologies Ltd.

Class	Property/Method	Description/Notes
	`attributes`	Returns any attributes associated with the current node, to define the **NamedNodeMap [20]**. If there are no attributes, then an empty `NamedNodeMap` is returned.
	Methods	
	`getElementsByTagName(tag)` `getElementsByTagName(namespace, tag)`	Returns a `NodeList` (or **HTMLCollection [19]**) of all element nodes for which the `TagName` matches the `tag` parameter provided, and for which the `namespace` matches the `namespace` parameter provided. If the `namespace` parameter is not provided, then the `namespace` syntax is ignored and need not be specified as part of the supplied `tag` parameter. If there are no matches, an empty `NodeList` is returned.
	`hasAttribute(key)` [Boolean]	If the current node has the attribute name specified in the `key` parameter, this method returns `true`. Otherwise, it returns `false`.
	`getAttribute(key)` [string]	Retrieves the attribute which has the name specified in the `key` parameter. If the attribute does not exist, a value of `null` is returned.
NamedNodeMap		This contains a collection of attributes obtained from a node as key/value pairs. It is usually accessed using the array access operator, with its contents being accessed via either the index or `key` name. The SDK only accesses the `NamedNodeMap` via the numeric index.
	Properties	
	`length` [number]	The number of key/value pairs contained in the current collection.
	`[ ]`	Allows access to an individual key/value pair based either on its numeric index or Its `key` name. Although the SDK uses this method, it is possible to convert all calls to access via the **item(index) [20]** method instead.
	Methods	
	`item(index)`	Retrieves the **attribute object [20]** from the specified `index` position within the collection. Index values are zero-based. If the collection does not contain the requested index, a `null` object is returned.
Attr		This is the attribute object, represented by a single `key/value` attribute pair.
	Properties	
	`name`	The attribute `name` that represents the `key` of the `key/value pair`.
	`value`	The attribute `value` of the `key/value` pair.

Copyright © 2009-2018 Yospace Technologies Ltd.

7. Tracking Playback in the SDK

In order that the SDK can fire both the **timeline** and **non-timeline** beacons listed in **Supported VAST Tracking Beacons [29]** at the appropriate times, you need to track playback in the SDK as follows:

- For the **non-timeline** beacons to fire, you need to obtain notifications of the UI and other changes referred to in **Reporting Player States [21]**.
- For the **timeline** beacons to fire, you need to interpret the metadata tags that are carried in the encoded ads in Live streams (refer to **Interpreting and Delivering Metadata [26]**). In other forms of playback, it is the position of the playhead that permits the SDK to fire the **timeline** beacons at the appropriate times. The app needs to report this position via the playhead timer. For example:

```
var lastPosition = -1;
var positionPoller = setInterval(function() {
// Make call to video player API to obtain playhead position.
var newPosition = player.getCurrentPosition();
if (newPosition != lastPosition) {
lastPosition = newPosition;

// Report updated playhead position to the SDK. Parameter is the playback position in seconds.
sessMgr.reportPlayerEvent(YSPlayerEvents.POSITION, newPosition);
}
}, 250);
```

7.1. Reporting Player States

The instance implements the **YSPlayerEvents** interface to enable the client app to report various playback state changes to the SDK.

The states can be invoked using the `reportPlayerEvent` function, for example:

```
Function reportPlayerEvent(evt /* a YSPlayerEvents constant*/, data)
```

The `evt` parameter is an enumerated type that determines which event type is being reported. Depending on this event type, an additional `data` parameter allows relevant, associated information to be provided.

The client app must call each event at the appropriate time in order for the analytics to be reported correctly. However, please note that support for some of the events, for example `pause`, `resume`, `skip`, is dependent on **Playback Policy [38]**.

The following events should be reported, as a minimum:

 Copyright © 2009-2018 Yospace Technologies Ltd.

Event	Parameter Type	Description
`START`	`null`	The stream has started playing. This event **must** be reported at the start of the playback session (that is to say, before any other event type is reported), and should be reported each time that the player leaves its 'stopped' state. It is not necessary to report this event following a buffer or `pause` event.
`END`	`null`	The stream has finished playing. This event **must** be reported when the video player stops playing back a stream, either because it has reached the end of a piece of content, such as a fixed length VOD, or when the viewer stops playback. No further events should be reported following this event (except for `YSPlayerEvents.START`, which would indicate a new playback session).
`MUTE`	`bool`	The volume either has been muted (`true`), or is no longer muted (`false`).
`FULLSCREEN`	`bool`	The video player has either entered full-screen playback mode, or has left full-screen mode and returned to windowed playback: • `true` indicates that the player has entered full-screen mode; • `false` indicates that the player has left full-screen mode. This refers to the overall state of the player, and **not** the display/hiding of transport controls.
`POSITION`	`Number`	The playhead position has changed (since the last report). The parameter reported should be the current playhead position (in seconds) relative to the start of the playback session (in the case of Live), or relative to the start of the content (in the case of all other playback types)
`METADATA`		Metadata was extracted from the underlying content **and** the playhead position has reached (or passed) the presentation timestamp associated with the metadata. A detailed explanation about extracting metadata to can be found in **Interpreting and Delivering Metadata [26]**.
`PAUSE`	`null`	Playback was paused (intentionally) by the viewer.
`RESUME`	`null`	Playback was resumed (that is to say, 'unpaused') either intentionally (by the viewer) or otherwise.

Copyright © 2009-2018 Yospace Technologies Ltd.

Event	Parameter Type	Description
`SEEK_START`	`Number`	The viewer began a scrubbing operation to seek a new position. The parameter reported should be the playhead position (in seconds) relative to the start of the playback session (in the case of Live), or relative to the start of the content (in the case of all other playback types) at which the seek operation began.
`SEEK_END`	`Number`	The viewer completed a scrubbing operation to seek a new position. The parameter reported should be the new playhead position (in seconds) relative to the start of the playback session (in the case of Live), or relative to the start of the content (in the case of all other playback types).
`CLICK`	`null`	The viewer interacted with a *Linear Creative*, and followed the clickthrough. The app is responsible for opening the clickthrough URL, but the beacon is handled by the SDK.
`NONLINEAR`		The viewer interacted with a *Non-Linear Creative*, and followed the clickthrough. The app is responsible for opening the clickthrough URL, but the beacon is handled by the SDK. The parameter refers to the index of the item within the `NonLinears` array from the `VASTAd` instance (`sessMgr.session.currentAdvert.advert.nonLinears`).
`ERROR`	`null`	A playback error occurred and the video player was unable to resume playback. As a result, this is considered an unrecoverable (fatal) playback situation. No further events should be reported.

7.1.1. Handling Clickable Elements

There are two main types of creative element that may be presented by the client app:

- **Clickable Linear Creatives [24]**, which are a requirement.
- **Clickable Non-Linear Creatives [24]**, which are optional.

For both types of creative, it is the responsibility of the client app to detect user interaction (clicking/touching) with the creative, and then report this to the SDK. Additionally, in the case of **Non-Linear** creatives only, the client app is also responsible for their presentation to the viewer. This includes both the legacy VAST 2.0-compliant **Non-Linears**, and the more recent VAST 3.0 Icons.

 Copyright © 2009-2018 Yospace Technologies Ltd.

7.1.1.1. Clickable Linear Creatives

The following code fragment shows how to inspect various aspects that describe the contents of ***Linear Creatives***, as well as how to obtain the clickthrough target URL (which is the URL that should be opened by the client when the creative is clicked on by the viewer).

```
function AdvertStart(miid)
{
 var advert = sessMgr.session.currentAdvert;

        var rawAdvert = advert.advert;
        Console.WriteLine("Advert sequence is " + rawAdvert.sequence);

        var linear = rawAdvert.getLinear();
        var nonlinears = rawAdvert.getNonLinears();

        if (linear != null) {
            console.log("Advert has a linear creative");
            console.log("Clickthrough URL is: ", linear.getClickThrough());
            console.log("Ad duration: (seconds) " + linear.getDuration());

            var medias = linear.getAllMedias();
            console.log("Advert had " + medias.length + " media creative URLs");

            for (var i = 0; i < medias.length; i++)
 {
            var media = medias[i];
            console.log("Media creative :" + i);
            console.log("Media URL: " + media["src"]);
            for (var attrib in media)
            {
                   // This will print out all attributes of the creative
                   // e.g. delivery, type, width, height, codec, id etc.
                   // The actual URL of the media file will be contained in
                   // an attribute with the key "src"
                   if (media.hasOwnProperty(attrib)) {
                    console.log(" " + attrib + " = " + media[attrib]);
             }
         }
     }
 }
```

Typically, the app will place a transparent overlay on top of the video element, or intercept click/touch events so that it can report the click event to the SDK.

Having opened the clickthrough target URL, the app reports the click event to the SDK, as follows:

```
function handleLinearClicked() {
      sessMgr.ReportPlayerEvent(YSPlayerEvents.CLICK, null);
 }
```

7.1.1.2. Clickable Non-Linear Creatives

The full list of available ***Non-Linear Creatives*** is shown in the following code fragment:

```
function AdvertStart(miid)
{
        var advert = sessMgr.session.currentAdvert;
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
        var rawAdvert = advert.advert;
        console.log("Advert sequence is " + rawAdvert.sequence);

        var linear = rawAdvert.getLinear();
        var nonlinears = rawAdvert.getNonLinears();
        var icons = linear?.Icons;

        if (nonlinears != null) {
        console.log("Advert has " + nonlinears.length + " non-linear creatives");

        for (var i = 0; i < nonlinears.length; i++) {
             console.log("Non-Linear creative index :" + i);
             console.log("Clickthrough URL: " + nonlinear.getClickThrough());

             var resource = nonlinear.getAllResources();
             console.log("Non-Linear HTML Asset: " + resource.html);
             console.log("Non-Linear IFrame Asset: " + resource.iframe);

             Var images = resource.images;

             for (var image in images)
          {
          // This will print out all static images in the Non-Linear by mime type.
          if (images.hasOwnProperty(image)) {
                    console.log(" Mime Type '" + image + "' URL: " + images[image]);
                }
             }
      }
     }
}
```

A similar approach can be used to iterate and traverse the **Icons** property. Handling a click on the image would then be done in the following fashion:

```
function handleNonLinearClicked(index) {
// Note that the index which is used here should match the index
// shown in the previous example.
/* If you prefer to use the instance of the VASTNonLinear instead,
you could retrieve the index using code similar to the following
(assuming the supplied instance was called "nonLinear"):
var advert = sessMgr.session.currentAdvert;
if (advert != null) {
varrawAdvert = advert.advert;
var index = -1;
var nonlinears = rawAdvert.getNonLinears();
for (var i = 0; i < nonlinears.length; i++) {
 if (nonlinears[i] === nonLinear) {
 index = i
 Break; }
}
*/

if (index >= 0) {
 sessMgr.ReportPlayerEvent(YSPlayerEvents.NONLINEAR, index);
 }
}
```

7.1.2. Pause and Resume for Live Pause

A Live Pause-enabled stream can be configured on the back end to be paused for up to 90 minutes. When a viewer pauses and resumes such a stream, you must ensure that the appropriate notifications are raised by your video player implementation:

 Copyright © 2009-2018 Yospace Technologies Ltd.

This section does not apply to either ***Video On Demand*** or ***NLSO***.

- For Pause, report `YSPlayerEvents.PAUSE`.
- For Resume, report `YSPlayerEvents.RESUME`.

```
function - pause()
    {
       this.videoPlayer.pause();
      sessMgr.reportPlayerEvent(YSPlayerEvents.PAUSE);
```

While paused, the SDK polls the Yospace CSM to maintain the client session and a growing playlist for the player. The SDK also polls the VMAP in order to deliver a regularly updated timeline to the app.

7.2. Interpreting and Delivering Metadata

The metadata is used to report the current playhead position when ads are playing in a Live stream. It is normally carried within an ***ID3 tag*** (refer to **ID3 Timed Metadata for HLS and MPEG-TS Streams [26]**), or an EMSG ISO box (refer to **EMSG in MPEG-DASH [27]**). Typically, HLS uses ID3, and MPEG-DASH uses EMSG. The actual content of the metadata will be the same in both cases.

Video segments using MPEG-TS format use the ID3 mechanism for timed metadata (refer to **ID3 Timed Metadata for HLS and MPEG-TS Streams [26]**). Segments in the fragmented MP4 format (used by both MPEG-DASH streams and HLS+CMAF streams) use EMSG (refer to **EMSG in MPEG-DASH [27]**).

Besides the key parts that make up the content of the metadata, there is also a part that represents the timing information of the metadata (refer to **Important Note about PTS Timestamps [28]**).

7.2.1. ID3 Timed Metadata for HLS and MPEG-TS Streams

In order to track the playhead in Live steams, Yospace's SSAI relies on the ID3 metadata being carried in the encoded ads that are inserted into the HLS stream.

HLS streams are delivered as MPEG Transport Streams in ***.ts*** segments. Within each of these segments, the data is structured as a sequence of 188-byte long Transport Stream packets. Each packet has a Packet Identifier code (or PID) that defines the nature of the content stored in that packet.

Once the Transport Stream packets for a given stream have been processed by the player, they are assembled into a sequence of variable-length Packetised Elementary Stream (PES) packets.

The payload is structured according to the ID3 standard, a copy of which can be found here:

`http://id3.org/id3v2.4.0-structure`

Referring to section 3.1 of the above standard, the data begins with a specific header format that occupies the first 10 bytes of the data. There is no extended header. What then follows is a series of ID3 frames (see section 4 of the standard). In a Yospace stream, these frames have the following Frame ID values:

 Copyright © 2009-2018 Yospace Technologies Ltd.

Frame	Description
YMID	Contains the `Media ID` of the ad.
YTYP	Contains the type of metadata ('S', 'M', 'E' for Start, Middle, End, respectively).
YDUR	The offset/duration from the beginning of the segment that contains the metadata.
YSEQ	The sequence number of this segment (represented as 'N:T', where 'N' is the segment number and 'T' is the total count of segments in this ad - for example, '1:5' means segment 1 out of 5).
YSCP	A customer-specific identifier that is typically unused and may be ignored.

For each of the preceding frames, the content associated with the frame is a simple string that starts with an `0x03` byte, to indicate that the string value is in UTF-8 format, and then contains a `0x00`-terminated string.

When decoding the frames for reporting to the SDK, the `0x03` prefix should be skipped.

7.2.2. EMSG in MPEG-DASH

MPEG-DASH streams are usually delivered as ISO Base Media File Format (ISO-BMFF)-type MP4 content. This format presents various parts of the content (headers, metadata and stream data) contained within 'boxes'. Each box has a 4-character identifier (its ATOM identifier), as well as its size, additional properties and payload.

For a Yospace stream, the metadata can be found in a series of 'EMSG'-type boxes, which are often enclosed at the beginning of the MP4 fragment. Each EMSG box payload contains the metadata as a string of key/value pairs. An example EMSG value is shown below:

```
YMID=525576649,YSCP=525576649,YSEQ=1:2,YTYP=S,YDUR=0
```

The `scheme_id_uri` field for Yospace EMSGs has the following string value: `urn:yospace:a:id3:2016`. Only EMSGs with this `scheme_id` should be reported to the SDK.

The list of key/value pairs is comma-separated and the data represented in each pair is equivalent to the same ID3 frame in HLS (refer to **ID3 Timed Metadata for HLS and MPEG-TS Streams [26]**).

Your app receives EMSGs, repackages them, and delivers them as timed metadata to the SDK as it receives them. However, the player sometimes delivers EMSGs in an unexpected order at the boundary between ads. In order to work around this behaviour, it is necessary for the app to condition the delivery of metadata to the SDK in accordance with the following logic:

- The EMSG data for an ad contains the following fields:
  - `YTYP`: this is a character representing the position of the EMSG within a fragment and can be any of `S` (Start), `M` (Middle) or `E` (End).
  - `YSEQ`: this is a ‘string’ of format `n:m` where `n` is the fragment number and `m` is the number of fragments in an ad.

 Copyright © 2009-2018 Yospace Technologies Ltd.

The player may deliver the final EMSG of an ad (`E m:m`) after the first EMSG of a subsequent ad (`S 1:m`). In this instance, the app must **not** deliver the `(E m:m)` to the SDK: the delivery of `(S 1:m)` on its own will cause the SDK state machine to end the previous ad and start the next one.

7.2.3. Delivery of Metadata to the SDK

Having extracted and parsed the metadata from its underlying storage mechanism, the client app will now have (for ad segments) a number of metadata objects.

The SDK expects each of the frames for a given piece of metadata to be grouped together into a single object.

An example `Metadata` object is shown below:

```
var metadata = {
 YMID: "12345678",
 YSEQ: “1:5”,
 YTYP: “S”,
 YDUR: “0.0”,
 YSCP: “12345678”
};
```

The created object should be reported to the SDK using the `ReportPlayerEvent()` method, using the `METADATA`-enumerated type value and the instance as the parameter:

```
sessMgr.reportPlayerEvent(YSPlayerEvents.METADATA, tag);
```

7.2.3.1. Important Note about Presentation Time Stamps

In the case of PES-based metadata delivered by HLS streams, the PES packet will contain a Presentation Time Stamp (PTS) for the packet, which indicates the exact playhead position at which the metadata should be reported.

In the case of EMSG-based metadata delivered by MPEG-DASH streams, the **EMSG** box contains properties that collectively indicate the exact playhead position at which the metadata should be reported. The `presentation_time_delta` field gives the time offset from the beginning of the fragment represented in `timescale` units (where the value of `timescale` represents one second). For example, for a `timescale` of `90000`, a `presentation_time_delta` of `180000` indicates 2.0 seconds from the start of the fragment.

Some video players may choose to report all metadata to the client application at the point at which they are decoded from the source fragment. This means that one or more metadata blocks are reported to the client at exactly (or almost) the same time, and in any case much earlier than they should be.

Reporting the metadata blocks to the SDK at the point of decode will prevent the correct operation of the SDK. You **must** therefore defer reporting them to the SDK until the correct playhead position has been reached. This may often involve setting up an event to fire at some point in the future. If the playback of the stream is paused either intentionally or due to playback buffering, these future events need to be rescheduled according to the length of the pause in the playback.

 Copyright © 2009-2018 Yospace Technologies Ltd.

7.3. Supported VAST Tracking Beacons

The following table lists the VAST tracking beacons that are supported by the Ad Management SDK. They are split into the **timeline** variety that require interpretation of the metadata tags carried in the encoded ads in Live streams (refer to **Interpreting and Delivering Metadata [26]**), and the **non-timeline** ones that are dependent on **Reporting Player States [21]**. The table also describes the conditions under which the beacons are fired in both Live and VOD playback.

The **timeline** beacons are fired in the following order:

1. `Impression`
2. `creativeView` (Linear)
3. `creativeView` (Non-Linear)
4. Quartiles (`start`, `firstQuartile`, `midpoint`, `thirdQuartile`, `complete`) + `progress`.

VAST Beacon	Live Behaviour	VOD Behaviour
Timeline		
`impression`	The SDK fires all defined `impression` URIs as soon as it receives the first metadata tag in the first segment of the ad.	The SDK fires all defined `impression` URIs as soon as the playhead is reported to have transitioned from content to an ad, or from one ad to a different ad.
`creativeView`	The SDK fires all defined `creativeView` event URIs as soon as it receives the first metadata tag in the first segment of the ad, and after it has fired any `impression` URIs.	The SDK fires all defined `creativeView` event URIs as soon as the playhead is reported to have moved beyond the first segment of an ad, and after firing any `impression` URIs.
`start` (not supported for ***Non-Linear Creatives***)	The SDK fires all defined `start` URIs as soon as it receives the first metadata tag in the first segment of the ad, and after it has fired any `creativeView` URIs.	The SDK fires all defined `start` URIs as soon as the playhead is reported to have moved beyond the first segment of the ad, and after the SDK has fired any *creativeView* URIs.
`firstQuartile` (not supported for ***Non-Linear Creatives***)	The SDK fires all defined `firstQuartile` event URIs as soon as it receives the first metadata tag after 25% of the ad has been played.	The SDK fires all defined `firstQuartile` event URIs as soon as the playhead is reported to have moved beyond the quarter point of the ad.
`midpoint` (not supported for ***Non-Linear Creatives***)	The SDK fires all defined `midpoint` event URIs as soon as it receives the first metadata tag after 50% of the ad has been played.	The SDK fires all defined `midpoint` event URIs as soon as the playhead is reported to have moved beyond the mid-point of the ad.
`thirdQuartile` (not supported for ***Non-Linear Creatives***)	The SDK fires all defined `thirdQuartile` event URIs as soon as it receives the first metadata tag after 75% of the ad has played.	The SDK fires all defined `thirdQuartile` event URIs as soon as the playhead is reported to have moved beyond the ad's 75% mark.
`complete` (not supported for ***Non-Linear Creatives***)	The SDK fires all defined complete event URIs as soon as it receives the final metadata tag in the last segment of the ad.	The SDK fires all defined complete event URIs as soon as the playhead is reported to have moved beyond the final segment of the ad.
`progress` (not supported for ***Non-Linear Creatives***)	The SDK fires all defined progress event URIs as soon as it receives the first metadata tag after the offset of the ad has been played.	The SDK fires all defined progress event URIs as soon as the playhead is reported to have been offset after moving beyond the first segment of an ad.
Non-Timeline		
`mute`/`unmute` (not supported for ***Non-Linear Creatives***)	The viewer activated the mute control and muted/unmuted one of the ***Linear Creatives***. The SDK fires each defined `mute` or `unmute` event URIs as soon as it receives an associated notification via **Reporting Player States [21]**.	

Copyright © 2009-2018 Yospace Technologies Ltd.

<table>
<tr><th>VAST Beacon</th><th>Live Behaviour</th><th>VOD Behaviour</th></tr>
<tr><td><code>fullscreen</code>/
<code>exitFullscreen</code></td><td colspan="2">The viewer activated a control to extend the ad to the edges of the viewer's screen, or to reduce the ad to its original dimensions.

These events are used to track the player extending to, and exiting from, full screen, and not the Linear Creatives and Non-Linear Creatives.

The SDK fires all defined <code>fullscreen</code> or <code>exitFullscreen</code> event URIs as soon as it receives an associated notification via Reporting Player States [21].</td></tr>
<tr><td><code>pause</code>/<code>resume</code> (not supported for Non-Linear Creatives)</td><td colspan="2">The viewer activated the pause control and paused/unpaused one of the Linear Creatives.

The SDK fires all defined <code>pause</code> or <code>resume</code> event URIs as soon as it receives an associated notification via Reporting Player States [21].</td></tr>
<tr><td><code>rewind</code> (not supported for Non-Linear Creatives)</td><td colspan="2">The viewer activated the rewind control to access a previous point in one of the Linear Creatives.

The SDK fires each defined <code>rewind</code> event URI as soon as it receives an associated notification via Reporting Player States [21].</td></tr>
<tr><td><code>closeLinear</code> (not supported for Non-Linear Creatives)</td><td colspan="2">The viewer clicked a close control on one of the Linear Creatives.

The SDK fires all defined <code>closeLinear</code> event URIs if the viewer closes the ad, as soon as it receives an associated notification via Reporting Player States [21].</td></tr>
<tr><td><code>skip</code> (not supported for Non-Linear Creatives)</td><td colspan="2">The viewer activated a control to skip one of the Linear Creatives, this being a different control from the one that might be used with <code>closeLinear</code>.

The SDK fires all defined <code>skip</code> events URIs as soon as it receives an associated notification via Reporting Player States [21].</td></tr>
<tr><td><code>close</code></td><td colspan="2">The viewer clicked the Close button on one of the Non-Linear Creatives.

The SDK fires all defined <code>close</code> event URIs as soon as it receives an associated notification via Reporting Player States [21].</td></tr>
</table>

 Copyright © 2009-2018 Yospace Technologies Ltd.

8. Handling Analytic Callbacks from the SDK

The SDK will raise callbacks at certain predetermined points so that the client app can handle entering and leaving ads appropriately (for example, updating on-screen timers or presenting a clickable cover layer). There are three main categories of callback function:

Category	Description
Ad Transitions (refer to **Content and Ad Transition Callbacks [31]**).	Invoked to show when ads and their respective breaks start and end.
Timeline Information (refer to **Timeline Information Callbacks [32]**).	Invoked when changes are received to the current playback timeline.
Other Callbacks (refer to **Other Callbacks [36]**).	Informational callbacks that are provided as a convenience to the client.

8.1. Content and Ad Transition Callbacks

Shown below is a table of callback functions that are invoked, together with the associated parameter types and descriptions:

Callback function (registered using `registerPlayer` on the `YSSessionManager` instance)	Parameter Type	Description
`AdBreakStart`	`YSAdBreak`	An ad break has started playing. The break will contain one or more ad items (each of which may be filler slate as opposed to actual ads). Typically, this function may be used to present on-screen messages to notify the viewer that they are watching an ad, or possibly to modify the availability of transport controls in the video player to prevent scrubbing of the ad content (for VOD-type content). The parameter contains the **`YSAdBreak`** instance of the break that is just starting.
`AdBreakEnd`	`YSAdBreak`	The previously started ad break has ended. There are no more ads left to be played in the break, and the assumption is that the stream has returned to playing the original content. Typically, this event is used to remove any previously presented on-screen messages, or to re-enable transport controls that were previously disabled. The parameter contains the **`YSAdBreak`** instance of the break that has just ended.

 Copyright © 2009-2018 Yospace Technologies Ltd.

Callback function (registered using `registerPlayer` on the `YSSessionManager` instance)	Parameter Type	Description
`AdvertStart`	`string`	An individual ad within a break has started playing. The string is the (Yospace) Media Item ID of the ad being played. This callback function, together with `AdvertEnd` (described next), is provided just as a convenience, and there are no particular actions that need to be taken. This might provide an opportunity to display or remove any non-linear type ads, but theoretically it would also be possible to use these functions in order to advance an ad counter displayed on top of the video (although this is restricted to VOD and NLSO, and would not be possible for Live streams).
`AdvertEnd`	`string`	An individual ad within a break has completed playback. The string is the (Yospace) Media Item ID of the ad that has just finished playing.

8.2. Timeline Information Callbacks

The timeline callback function is used to report changes to the current playback timeline to the client application. This might happen, for example, as a result of the content at the live end of the playlist being extended (as new content becomes available), or because a new ad break has been added at the live end of the playlist.

Callback function (registered using `registerPlayer` on the `YSSessionManager` instance)	Parameter Type	Description
`TimelineUpdated`	`YSTimeline`	The current timeline representation has been updated. The parameter object attached to this event contains a reference to the current `YSTimeline` instance. The reported timeline does not identify which changes were made. It is the responsibility of the client application to maintain a deep copy of the previous timeline instance if this is to be detected. In most cases, however, it is not required.

8.2.1. Working with the Timeline

For Non-Linear stream types, the SDK maintains a representation of the content timeline for the stream. This enables the client app to determine the layout of the stream by content type (**Ad**, **VOD** and **Live**).

 Copyright © 2009-2018 Yospace Technologies Ltd.

Each applicable session type contains a property that, for the sake of convenience, can be accessed directly from the `YSSessionManager` instance as the `Timeline` property.

The `Timeline` instance only has a few properties that are of interest to the client app, these being:

Property /Method	Type	Description
`getTotalDuration()`		The total playback duration of all items contained within the `Timeline` instance (in seconds). This does not include any 'expired' content (for example, content that has been removed from the start of a Live Pause session due to the maximum DVR window size being reached).
`startOffset`		The playhead offset of the start of the first item in the timeline. Usually this will be zero, but, in the case of Live Pause sessions that have reached their DVR window, this will be non-zero in order to indicate the duration of the content that has expired.
	`bool`	Whenever a change is made to the timeline instance, a `TimelineUpdated` callback function is invoked in the client app. At any point, however, the client may request the current timeline and call the `isModified()` method to determine whether it has changed since the last time this method was called. Reading this property will reset its value to `false` and it will remain as `false` until any further updates are made.
`getAllElements()`	`Array /* of YSTimelineElement */`	Each period of time represented within the `Timeline` instance is encapsulated by a single `YSTimelineElement` instance. This property allows all such instances to be obtained by the client app.

Refer also to **Updating the Timeline [33]** and **Constructing a Dynamic Timeline for Live Pause [34]**.

8.2.1.1. Implementing UI Scrub Control

You may wish to show the timeline on the UI as a scrubber bar, so that the viewer can seek to different parts of the stream. You may, for example, wish to show the location of an ad break using a coloured bar on the timeline. An example scrubber bar is provided with the example app, which displays the ad breaks in one colour then 'dims' the coloured bar after the ad break has been watched.

8.2.1.2. Updating the Timeline

When playing back Non-Linear streams, the function `UpdateTimelinetimelineUpdateReceived:` in the Player Callbacks object will be invoked to provide notification that additional timeline data is available from the CSM.

The payload is the current `YSTimeline` instance, which you can use directly.

 Copyright © 2009-2018 Yospace Technologies Ltd.

However, a simpler method is to retrieve the entire timeline from the Yospace stream, as follows:

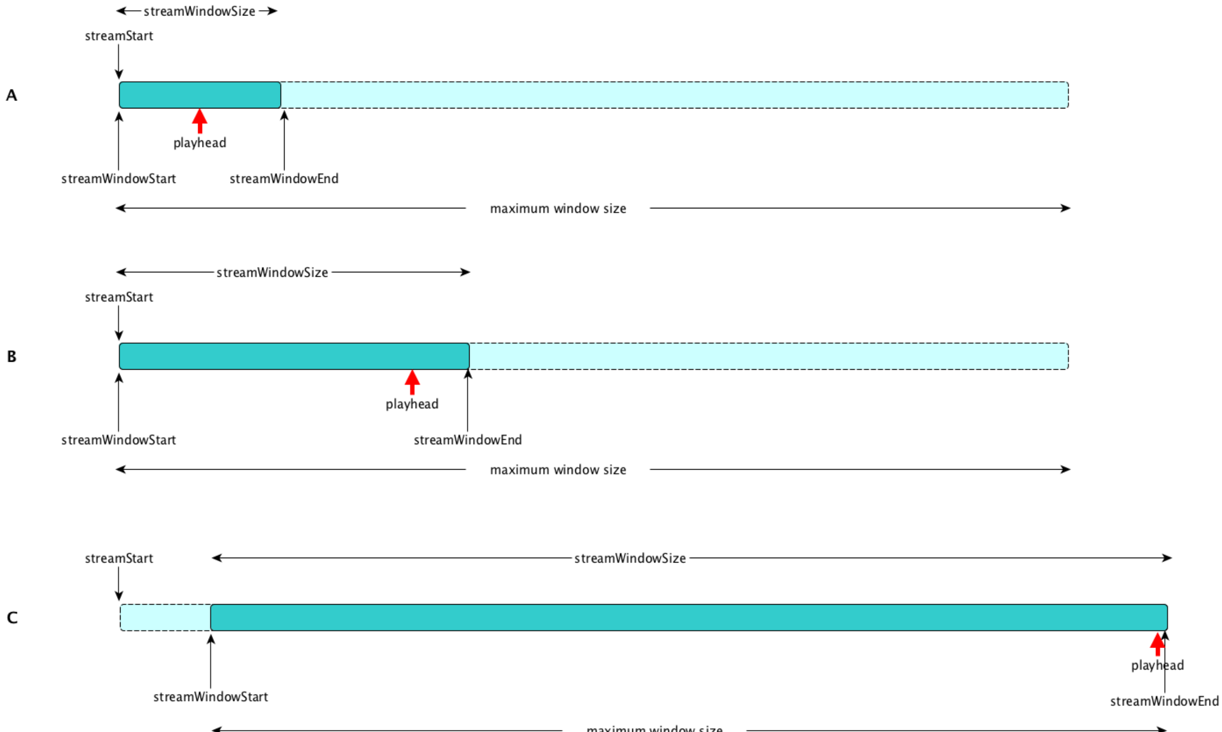
```
var cb_obj = {
    UpdateTimeline: function(timeline) {
        // work with timeline here
    }
};

// or from any function:

var timeline = sessMgr.getTimeline();
// timeline can now be used here
```

8.2.1.2.1. Constructing a Dynamic Timeline for Live Pause

The Yospace stream object contains properties that you can use to construct a dynamic timeline for Live Pause.

This section does not apply to either ***Video On Demand*** or ***NLSO***.

The following diagram shows how the properties relate to a theoretical dynamic timeline:



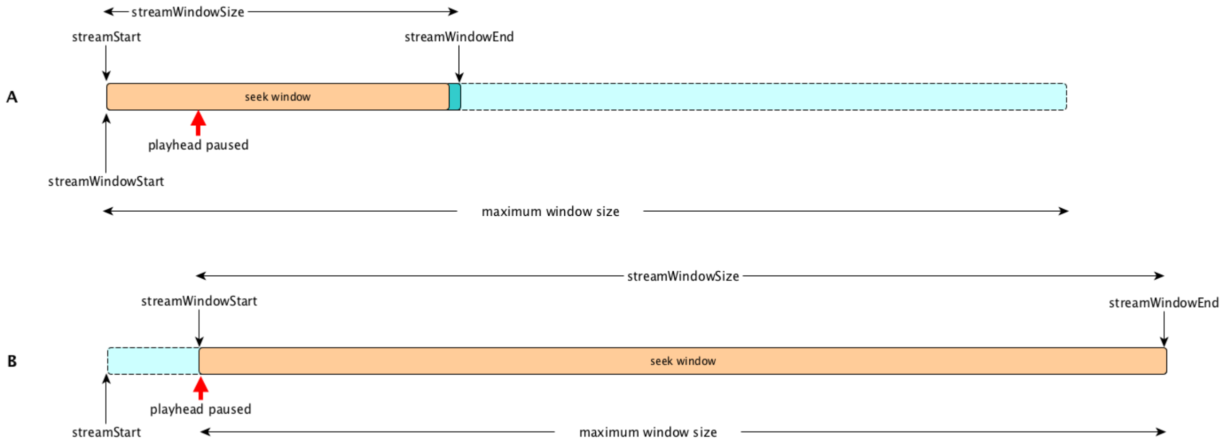
A. Stream startup - the stream window size is equal to the number of segments in the level playlist * their durations. The stream window start is equal to the stream start. The maximum window size, configured for that stream, is much greater than the window size. The playhead, according to typical player behaviour, is 3 segments from the window end.

B. In-stream playback - the stream window size has grown since more segments have been added to the level playlist. The stream window start is still equal to the stream start. The maximum window size has not yet been reached. The playhead is still 3 segments from the window end.

 Copyright © 2009-2018 Yospace Technologies Ltd.

C. The Window is now moving - the stream window has grown to the maximum size configured for the stream. The stream window start now moves away from the stream start, maintaining the maximum window size. The playhead is still 3 segments from the window end.

The following diagram shows pausing of the playhead in this theoretical timeline:



A. The window size is smaller than the maximum value configured. The playhead has been paused for a time. Therefore, there is a 'seek' window growing ahead of the paused point (as well as the 'seek' window **start**, for which the value is when playback started).

B. The window size has reached the maximum value and the window has moved on so that the window start is now at the point where playback was paused. You will need to make a decision on the behaviour of the application - otherwise, when playback is resumed, the stream will either jump or fail. The recommendation is to resume playback automatically once the maximum window size has been reached, as for Live Pause.

8.3. Inspecting the Break currently playing

During stream playback, ads are grouped into a single break. In the case of playing back a Live stream, only the break that is currently playing can be inspected by the client app - once the break has played out, the current break will be non-existent (and reported as 'null' from any API calls).

For other stream types, the break persists for as long as the break's media fragments remain available. For non-Live streams, whilst the current break will be reported as 'null' when the player is not currently viewing an ad, it is still possible to inspect both the breaks that have elapsed and those that are to occur in the future (and have already been resolved).

The ideal time to obtain the break information is when `AdBreakStart()` is invoked (refer to the **Content and Ad Transition Callbacks [31]**).

The information can be obtained as follows:

```
function AdBreakStart()
{
    var adBreak = sessMgr.session.currentBreak;
}
```

Having obtained the break, it is then possible to read its duration (`adBreak.getDuration()`) and to obtain the list of ads contained within the break (`adBreak.adverts`).

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
function AdBreakStart()
{
 var adBreak = sessMgr.session.currentBreak;
           console.log("Advert break has total duration: " +
            adBreak.getDuration() + " seconds");

        var adverts = adBreak.adverts;
           console.log("Break contains " + adverts.length
             + " adverts");

        for (var i = 0; i < adverts.length; i++) {
           console.log("Advert " + adverts[i].advert.id +
          " has duration: " + adverts[i].duration + " seconds");
 }
}
```

8.4. Inspecting the Ad currently playing

When an ad is currently being played in the stream, it is possible to obtain specific, associated information. The ideal time to obtain this information is when `AdvertStart()` is invoked (refer to the **Content and Ad Transition Callbacks [31]**), because this notifies the client that an ad has started to play.

A detailed fragment is shown below illustrating how to inspect various parts of the ad hierarchy:

```
var AdvertStart(miid){
 var advert = sessMgr.session.currentAdvert;

 var rawAdvert = advert.advert;
 console.log("Advert sequence is " + rawAdvert.sequence);

 var linear = rawAdvert.getLinear();
 var nonlinears = rawAdvert.getNonLinears();

 if (linear != null) {
 console.log("Advert has a linear creative");
 console.log("Clickthrough URL is: " + linear.getClickThrough());
 console.log("Ad duration: (seconds) : " + linear.getDuration());
 }
}
```

The hierarchical lineage of an ad is presented to your app for use with a **Viewability Adapter [48]**. The purpose of this is to collate any IDs that are presented while the CSM is following a chain of third party wrapper calls, so that the IDs can subsequently be extracted. For instructions on how to extract the IDs, refer to **Viewability Preparation and Integration [48]**.

8.5. Other Callbacks

The remaining callback functions are intended to provide state information back to the client application:

Copyright © 2009-2018 Yospace Technologies Ltd.

Callback function (registered using `registerPlayer` on the `YSSessionManager` instance)	**Parameter Type**	**Description**
`AnalyticsFired`	`object`	A tracking beacon has been fired by the SDK. There is no specific action required by the client application. However, this information might be used in order to make additional tracking calls if required.

Copyright © 2009-2018 Yospace Technologies Ltd.

9. Playback Policy

As has been previously mentioned in **Configuring Playback Policy [10]**, the app queries the SDK to verify playback policy. It does this through the **YSPlayerPolicy** interface.

You might deploy the interface as follows:

```
<script type="text/javascript" src="yo-ad-management.min.js"></script>
<script type="text/javascript" src="YSPlayerPolicy.js"></script>
```

There can only ever be a single definition of the `YSPlayerPolicy` interface set up, whether this is the default sample version that came with the SDK, or your own modified version of it. Therefore, if you have created your own version, ensure that this is deployed instead of the original version supplied with the SDK.

To use the policy functions, the client must first obtain the policy reference from the SDK:

```
var policy = sessMgr.session.getPolicy();
```

You can then enable the app to make calls against the policy.

There are basically three types of callback:

- The Boolean types that return `true` or `false` to the app, for example `canPause`, `canRewind` and `canSeek`.

  For example, to call `canSeek` for a Non-Linear stream:

  ```
  var isSeekable = policy.canSeek();
  ```

- The `willSeekTo` method that returns to the app the position of the playhead to which a seek is allowed (based on a desired seek position).

  For example:

  ```
  // This function is called as a result of scrubbing in the player.
  // The target parameter is the requested seek position (in seconds).
  // How you implement this will depend on the player you are using
     function onSeekEnd(target) {
      var seekTo = policy.willSeekTo(target);
      if (seekTo !== target) {
          // This seek is NOT permitted. You may
          // need to issue a command to the player to return
          // to the closest permitted position (which is now in
          // the target variable).
          }
  }
  ```

- The `canSkip` method that returns to the app the delay in seconds before the ad can be skipped (otherwise, `-1`).

  For example:

  ```
  var timeToSkip = policy.canSkip();
  if (timeToSkip === -1) {
      // This ad cannot be skipped. Ever.
  } else if (timeToSkip === 0) {
      // Display “Skip Ad” button
  } else {
  ```

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
    // Display “Skip ad in x seconds”
    // where x is timeToSkip
}
```

Refer also to **Configuring Playback Policy [10]**.

9.1. Policy for Linear Playback

By default, `canRewind`, `canSeek` and `canPause` are set to `FALSE` for Live playback, and `canSkip` is set to `-1`. Although it is possible to modify the defaults for these methods, such modifications are not recommended for Live streams, and you should use **Live Pause Playback [7]** instead.

9.2. Policy for Non-Linear Playback

Of particular importance for Non-Linear streams is the use of the playback policy to define if and when a viewer must watch an ad break before being able to watch content.

There is a sample app implementation that defines a 'right to watch' policy (refer to **Configuring Playback Policy [10]**). The implications of 'right to watch' are that:

- If a viewer scrubs over an active ad break, the start position of that break is returned, and the calling app **must** honour seeking to that position.
- If the viewer scrubs over multiple active ad breaks, the start position of the closest active break is returned, and the viewer needs to watch only the ad break immediately prior to the content to which they have tried to seek, and not any before that.
- Once a break has been watched through to completion, it is set as 'inactive', which means that it can be scrubbed over freely.

9.3. Skip Offset

Policy handling supports the `skipOffset` property for Linear creatives in the Non-Linear playback modes ***Video On Demand***, ***NLSO*** and ***Live Pause***. This property identifies when skip controls are made available to the viewer.

If `skipOffset` for a Linear creative is defined in the VAST, then the corresponding 'skip' event should also be defined in the VAST.

The property allows a 'skip' event to be generated by the application, and a tracking pixel to be fired when the 'skip' event is defined for the ad being skipped over. The 'skip' functionality should not be used for Live behaviour (as is also the case for Pause and Scrub, for example) - otherwise, the analytic behaviour will be undefined for the duration of the associated ad break.

 Copyright © 2009-2018 Yospace Technologies Ltd.

You should ensure that any playback policy that you implement is in line with the 'skip' behaviour implemented in the app (refer to `canSkip` in the `YSPlayerPolicy` protocol described in the).

The following default policy is defined in the policy handling implementation with regard to the different playback modes:

- For Live playback, `-1` is returned.
- For VOD playback, the value of the property (typically, in seconds) is returned.
- For NLSO and Live Pause playback, the value of the property is returned, unless the current playhead position plus the remaining duration of the ad would move the playhead past the 'live' position of the player, in which case `-1` is returned.

Copyright © 2009-2018 Yospace Technologies Ltd.

10. Suppressing Analytics

Analytics need to be suppressed in the following cases:

- To prevent duplicate analytics from being fired while VPAID content is being played. The suppressed tracking URLs that are returned from the analytics suppression call is dependent on when the VPAID unit is dismissed - refer to **VPAID Implementation Procedure [43]**.
- If your app has failed to meet Viewability requirements with regard to ads (refer to **Viewability Adapter [48]**). You should suppress the firing of analytics until such time as your app meets the requirements.

You should always stop analytics at the end of a session, to prevent possible resource leaks. Refer to **Cleaning Up [50]**.

 Copyright © 2009-2018 Yospace Technologies Ltd.

11. Implementing VPAID in Live and VOD Playback

The Yospace Ad Management SDK supports the ***Video Player Ad Interface Definition*** (VPAID) in both Linear and Non-Linear video streams.

For more information on VPAID compliance, please refer to **https://www.iab.com/?s=vpaid**.

VPAID support is implemented via an adapter that provides a simple interface, which, when integrated into an app, allows the app to handle the intricacies of overlaying VPAID content on a video. Following implementation of the adapter, any analytic beacons that are defined in a VAST document are fired automatically for the associated VPAID ads.

The CSM will always stitch in the underlying VPAID content. The duration of the stitched content will depend on whether the VAST document defines a second `MediaFile` element that is to be used where a VPAID asset cannot be rendered. If this element is defined, then the CSM stitches this asset: otherwise, a default 'VPAID filler' asset is stitched in.

Because VPAID content can be of arbitrary length, in that the viewer can interact with it or close it prior to completion, supporting such content risks degrading the viewer experience, particularly in Live playback. However, the impact can be lessened via the judicious use of filler assets, particularly with regard to the length and background colour. These filler assets are set up in the yospaceCDS Management Console, in **VPAID Settings** for the relevant Promotion(s), as follows:

- For Live playback, in order to allow the SDK to preserve the linear timeline, set up a filler asset that is of a duration which enables it to be stitched into the Live stream in place of a VPAID unit, for example 30 seconds.
- For VOD playback, in order to allow each ad break to be identified, set up a short filler asset that can be stitched into the VOD stream - this need be no longer than two seconds.

If you wish to ensure that the specified filler is used instead of a VPAID `mediafile` element, then preface the asset ID with a minus sign.

Depending on the player, video may decoded in hardware, with only one video being played at a time. This means that the app must pause the content if a VPAID video is played back, and that, when the VPAID closes, playback of the original content must be resumed.

In Live playback, if the viewer closes the VPAID unit early, the underlying media/filler will be visible. Conversely, if the viewer continues to interact with the unit for a longer time than its duration, then, when the video resumes, the app will seek to the end of the underlying media and playback will be behind the Live point.

Whenever the VPAID unit is closed In VOD playback, it is the responsibility of the client app to move the playhead to the end of the underlying content so that playback can resume.

In the sample app, the moving of the playhead is done in the `AdStopped` event (refer to **Events [46]**), where:

```
video.currentTime = lastPosition + sessMgr.session.currentAdvert.duration;
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

11.1. VPAID Implementation Procedure

As prerequisites, the app must:

- supply a minimum of two UI objects: an empty `<div>` to contain the VPAID static elements, and a clickable element (for example, a button) to be used for 'skip' functionality);
- supply a `YSSessionManager` instance (as returned from the factory method).

You can then proceed as follows:

1. Include the `VPAID-adapter.min.js` JavaScript module.
2. Construct an instance of the `VPAIDAdapter` class and attach a listener to it which will receive callbacks:

```
var vpaid_cb = {
 AdLoaded: function() {
 // The video decode pipeline should be stopped
 // here and detached from the video element
 },

 AdStopped: function() {
 // The video decode pipeline needs to be restarted
 // here and re-attached to the video element
 },

 // Additional VPAID callbacks to suit the application. If a callback
 // exists, it will be triggered at the appropriate time, otherwise the
 // adapter will silently accept that it is not supplied.

 // All callbacks defined in the VPAID spec are accessible here, and are
 // named as per the spec. See Spec:
 // https://www.iab.com/wp-content/uploads/2015/06/VPAID_2_0_Final_04-10-2012.pdf
// (Pages 44-57)
```

3. In the player callback `AdvertStart`, query the ad for a VPAID unit and retrieve its data as a `VASTInteractive` instance.
4. Having suppressed analytics for the duration of the viewer interaction, paused playback, and saved the current playhead position, a call is made to `loadAdUnit` on the Adapter:

```
if (sessMgr.session.currentAdvert.hasInteractiveUnit())
{
        sessMgr.session.suppressAnalytics(true);
        sessMgr.reportPlayerEvent(YSPlayerEvents.PAUSE);
        vpaidWrapper.loadAdUnit(
        sessMgr.session.currentAdvert.getInteractiveUnit()
        );
}
```

5. Implement 'skip' behaviour in the `Button` handler.
6. Implement click-through behaviour in the VPAID `AdClickThru` callback method.
7. Implement skip/expand functionality in the respective VPAID callback methods, for example:

```
var vpaid_cb = {
    AdSkippableStateChange: function(skippable) {
        var expanded = vpaidWrapper.getAdExpanded();
        if ((expanded) || (!skippable)) {
            // hide skip button
        } else {
            // set skip button visibility
            // according to skippable parameter
        }
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

```
    },

    AdExpandedChange: function(expanded) {
        var skippable = vpaidWrapper.getAdSkippableState();
        if ((expanded) || (!skippable)) {
            // hide skip button
        } else {
            // set skip button visibility
            // according to skippable parameter
        }
    },

    AdClickThru: function(url, id, playerHandles) {
        if (playerHandles) {
            // We are responsible for opening the click url ourselves
            window.open(url, “blank”);
        }
    }
```

8. In the VPAID `AdStopped` callback, re-enable analytics as necessary and re-attach the video decoder to the video element:

```
sessMgr.session.suppressAnalytics(false);
    // re-attach and resume content playback here!
```

11.1.1. Implementing the IAB VPAID Interface for JavaScript

The Yospace JavaScript implementation makes available all those **Methods [44]**, **Properties [45]** and **Events [46]** described in the AIB's VPAID **specification** that may be required by the client app.

In order to call the **Methods [44]**, it is first necessary to obtain a reference to the underlying VPAID container (of type `VPAIDUnit`) from the `VPAIDAdapter` instance. This is obtained by calling the `getVPAID()` method on the adapter, for example:

```
/* Assuming you previously constructed an adapter as follows:
var adapter = new VPAIDAdapter(vpaidOverlay, video, sessMgr); */
var container = adapter.getVPAID();
container.stopAd();    // Invoke the standard VPAID method “stopAd”
```

11.1.1.1. Methods

The available methods are as follows:

- `handshakeVersion(version)` [1]
- `initAd(width, height, viewMode, desiredBitrate, creativeData, environmentVars)` [1]
- `startAd()`
- `stopAd()`
- `resizeAd(width, height, viewMode)`
- `pauseAd()`
- `resumeAd()`

[1] These methods should almost certainly never be called by a client app directly as they are managed by the `VPAIDAdapter` instance.

 Copyright © 2009-2018 Yospace Technologies Ltd.

- `expandAd()`
- `collapseAd()`
- `skipAd()`
- `subscribe(fn, event, scope)`[2]
- `unsubscribe(fn, event)` [2]

11.1.1.2. Properties

Certain properties of the VPAID unit can also be queried (and occasionally set) by the client app.

For the JavaScript implementation, the ad provides a `getter`, and, if the property is settable, a `setter` function, for each of the properties. Property-specific `getter` and `setter` functions are used in order to access the ad properties.

All properties defined by the VPAID **specification** are made available as callable methods. These include the following:

- `setAdVolume(val)`
- `getAdVolume()`
- `getAdLinear()`
- `getAdWidth()`
- `getAdHeight()`
- `getAdExpanded()`
- `getAdSkippableState()`
- `getAdRemainingTime()`
- `getAdDuration()`
- `getAdCompanions()`
- `getAdIcons()`

An example code snippet is shown below:

```
/* Assuming you previously constructed an adapter as follows:
var adapter = new VPAIDAdapter(vpaidOverlay, video, sessMgr); */
var container = adapter.getVPAID();
var skippable = container.getAdSkippableState();   // Read the "AdSkippableState" property
```

[2] These methods are indeed available, but there is an alternative, preferred approach outlined in **Events [46]**.

 Copyright © 2009-2018 Yospace Technologies Ltd.

11.1.1.3. Events

All events defined in the VPAID **specification** are also made available. Whilst the traditional `subscribe/unsubscribe` methods defined in the specification, and referred to in **Methods [44]**), are available, an alternative, preferred approach is also available.

To use this approach, an object is created which contains functions for the events to which the app wishes to listen. This object is then registered with the adapter as follows:

```
/* Assuming you previously constructed an adapter as follows:
var adapter = new VPAIDAdapter(vpaidOverlay, video, sessMgr); */


var vpaid_cb = {
    AdLoaded: function() {
        // AdLoaded event will call this function
    },

    AdLog: function(msg) {
        // AdLog event will call this function, passing
        // the message parameter
    }

    // etc.
};
adapter.addListener(vpaid_cb);
```

As many callback objects as required can be registered (although it is recommended that the number is kept small so as to optimise performance). If a callback needs to be de-registered, it can be done so as follows:

```
adapter.removeListener(vpaid_cb);
```

For the sake of convenience, some of the callback functions provided in this way will also carry additional parameters which are NOT defined in the VPAID specification. A list of the differences is shown below:

VPAID event	Callback Prototype	Parameters	Notes
`AdLoaded`	()	none	
`AdStarted`	()	none	Automatically converted[1]
`AdStopped`	()	none	
`AdSkipped`	()	none	Automatically converted[1]
`AdSkippableStateChange`	(`skippable`)	`skippable` is the new boolean `skippable` state	
`AdSizeChange`	(`width`, `height`)	`width` and `height` are the new `width` and `height`, respectively.	
`AdLinearChange`	(`linear`)	`linear` is the boolean `getAdLinear`value	
`AdDurationChange` `AdRemainingTimeChange`	(`duration`, `remaining`)	`duration` is the total reported VPAID duration, whilst `remaining` is the reported remaining time.	
`AdVolumeChange`	(`val`)	`val` is the new volume level.	
`AdImpression`	()	none	

 Copyright © 2009-2018 Yospace Technologies Ltd.

VPAID event	Callback Prototype	Parameters	Notes
`AdVideoStart`	()	none	Automatically converted[1]
`AdVideoFirstQuartile`	()	none	Automatically converted[1]
`AdVideoMidpoint`	()	none	Automatically converted[1]
`AdVideoThirdQuartile`	()	none	Automatically converted[1]
`AdVideoComplete`	()	none	Automatically converted[1]
`AdClickThru`	(`url`, `id`, `playerHandles`)	`url` is the URL of the clickthrough target; `id` is the ad Id provided by the VPAID unit; `playerHandles` constitute a boolean which indicates whether or not the client app needs to handle (i.e. open) the target URL.	
`AdInteraction`	(`adid`)	`adid` is the ad Id provided by the VPAID unit.	
`AdUserAcceptInvitation`	()	none	Automatically converted[1]
`AdUserMinimize`	()	none	Automatically converted[1]
`AdUserClose`	()	none	Automatically converted[1]
`AdPaused`	()	none	Automatically converted[1]
`AdPlaying`	()	none	Automatically converted[1]
`AdLog`	(`msg`)	`msg` is the message being logged.	
`AdError`		`msg` is the error message being logged.	

[1] These events are automatically converted to VAST tracking beacons and fired by the SDK. There is therefore no probable need to listen to these events.

 Copyright © 2009-2018 Yospace Technologies Ltd.

12. Viewability Adapter

The SDK supports the implementation of ***Moat*** in Live, Live Pause, VOD and NLSO streams through the delivery of extension data in an ad's VAST document. The hierarchical lineage of the ads can be used to extract the required information - refer to **Viewability Preparation and Integration [48]**.

12.1. Viewability Preparation and Integration

To support integration with Moat and other Viewability Adapters, the hierarchical lineage of ads is presented to the app via an extension block containing the structure of the VAST wrappers. This is to facilitate the collation of any IDs that are presented during third party wrapper calls, so that the IDs can subsequently be extracted.

The parameter `yo.ah` needs to be set to `true` in the CSM playback URL, in order for this extension block to be enabled and the ad hierarchy to be presented. If the parameter is set to `false` or not set, then the wrapper information is not added to the VAST response.

Assuming that the hierarchical lineage of each ad is being presented to your app, you can insert blocks of code to extract the IDs.

The following code sample is taken from `AdvertStart()`. The sample takes account of the fact that the ads might be inline, and therefore not in a wrapper, in which case the IDs need to be extracted from the ads themselves. The sample can be inserted within the code given in **Inspecting the Ad currently playing [36]**.

```
/* Insert between 'rawAdvert.sequence);' and
     'var linear = rawAdvert.getLinear()' */

      /* BEGIN: Wrapper ID extraction */
     var lineage = rawAdvert.AdvertLineage;
      if (lineage) {
         while (lineage) {
            Debugger.print("Lineage item: " + lineage.AdSystem + " (" + lineage.AdId + ")");
            lineage = lineage.child;
         }
      } else {
         Debugger.print("Advert not in a wrapper. AdSystem: " + rawAdvert.AdSystem + ", ID: "
            + advert.getAdvertID());
      }
      /* END: Wrapper ID extraction */
 }
}
```

The first item retrieved will always refer to the topmost wrapper (that is to say, Parent or Grandparent) and the last will be the ad itself. The sample code above will print out the hierarchy in order of seniority, that is to say:

```
Lineage item: FirstPartyServer (Wrapper1)
Lineage item: ThirdPartyServer (Advert1)
```


Copyright © 2009-2018 Yospace Technologies Ltd.

13. VAST Macros

The SDK supports substitution of the following macros:

- `[CONTENTPLAYHEAD]`: replaced by the current time offset `HH:MM:SS.mmm` of the video content.
- `[CACHEBUSTING]`: replaced by a random 8-digit number.
- `[ASSETURI]`: replaced by the URI of the ad asset being played.

In the case of multiple media files defined in the VAST document, the SDK will substitute the URL of the content with the highest bitrate, in line with the Yospace back-end behaviour. If the bitrate attribute is not present in any entry, the URL will be the first entry.

Copyright © 2009-2018 Yospace Technologies Ltd.

14. Cleaning Up

When there is only one stream, and a `YSSessionManager` instance is created successfully, then the SDK automatically cleans up outstanding resources in order that its memory can be freed up at the earliest point. However, in any of the following cases, you should manually intervene in order to clean up:

- The SDK returns a `NOT_INITIALISED` or`NO_ANALYTICS` status (refer to **Session Status Notification [15]**). Shut down the SDK and `null` the Session Manager, for example:

```
sessMgr.shutdown();
sessMgr = null;        // To allow for garbage collection
```

- For a VOD asset, the SDK is initialised and the stream plays to the end. Once this has happened, ensure that the SDK is shut it down before a new asset is loaded, for example:

```
sessMgr.reportPlayerEvent(YSPlayerEvents.END);
sessMgr.shutdown();
```

In the event that the existing asset is unloaded and cannot be restarted, it is not **mandatory** to shut down the SDK, but it is nevertheless advisable.

- The SDK is initialised, the stream plays, but the viewer selects a different video to be played during playback. The SDK **must** be shut down before the next session is created, as in the following example:

```
sessMgr.shutdown();
sessMgr = YSSessionManager.createForNonLinear(newurl, null, function(state,
result) {
console.log("SessionManager is Ready. Result: " + state);
if (state === YSSessionResult.INITIALISED) {
```

The only exception to the above point is where the selection of a different video causes the current HTML page to be reloaded - in this case, the current page would be unloaded by the browser first and there would therefore be no need to shut down the SDK. In contrast, where the lifecycle of the app can continue across multiple playbacks without reloading the page, the SDK **must** be shut down manually whenever the playback session (or video) changes.

Following shutdown, the Session Manager instance (or any subclass) should not be used or referenced in any way, as its state is undefined.

 Copyright © 2009-2018 Yospace Technologies Ltd.

15. Running in Debug Mode

Running your app in debug mode is recommended when **Validating your App prior to Release [52]**, so that trace information is output to the console.

To set the `DEBUGGING` configuration parameter for the particular stream (refer to **Setting the Configuration Parameters [12]**), you would preface the code listed in **Code Fragment invoking a Factory Method [13]**, with the following:

```
// Create a session for playing back a live stream
var YSSessionProps = { DEBUGGING: true };
var sessMgr = YSSessionManager.createForLive
    ("http://csm-e.cds.yospace.com/stream.m3u8",
     YSSessionProps, function(state, result) {
        console.log("SessionManager is Ready. Result: " + state);
```

 Copyright © 2009-2018 Yospace Technologies Ltd.

16. Validating your App prior to Release

Prior to releasing your app, it is required that you provide Yospace with either of the following, so that we can ensure that your app has implemented the Ad Management SDK lifecycle correctly:

- A candidate release for inspection (provided that you have an account), and the console log generated when **Running in Debug Mode [51]**.
- A network log that captures the app first initialising a stream, and then shutting it down after playing through at least one ad break. You should also provide the console log.

The above steps are intended to ensure, for your own benefit, that chargeable units, such as video analytics and user concurrencies, are accurate when the app goes live.

Copyright © 2009-2018 Yospace Technologies Ltd.

17. Language Reference

The **Yospace Developer Portal** not only provides access to the searchable, online version of this user guide, but also provides access to the following items that are referenced earlier in the guide:

- **The Ad Management SDK class documentation**.
- **The sample app(s)**.

For further information, please contact **support@yospace.com**.

Copyright © 2009-2018 Yospace Technologies Ltd.

18. Glossary

Central Streaming Manager (CSM)	An edge network service for delivering personalised video streams.
Common Media Application Format (CMAF)	Whereas HLS is typically delivered with Transport Stream containers and MPEG-DASH with fragmented MP4 (`fMP4`), CMAF specifies an `fMP4`-compatible container format that allows media to be encoded and stored just once. It can then be delivered via both the HLS and MPEG-DASH streaming protocols, thereby reducing costs and simplifying overall workflow.
ID3 tag	This is metadata, carried in an MPEG transport stream, that is used to tag a specific location in the stream with information. The metadata uses the ID3 audio file data tagging format for conveying the information. Yospace uses ID3 tags in ingested advert content in order to convey information to the client-side, during Live playback, about the ad that is currently playing.
Linear Creatives	The Linear creatives associated with an ad are the linear video units that are obtained from the ad server call, ingested into the Yospace CSM, and then stitched into the stream for playback in place of the original underlying source content.
Live Pause	A form of ***Non-Linear*** playback that allows a Live stream to be paused for up to 90 minutes. On resumption, the stream plays back from the point at which it was paused, provided that this remains within the 90-minute window. If the stream is paused for more than 90 minutes, then the behaviour is defined by the app.
Moat	Oracle's digital measurement tool, supported by Yospace, in Live, VOD, NLSO and Live Pause streams, through delivery of extension data in an ad's VAST document. For more information, please refer to **https://moat.com/analytics**.
NLSO	Non-Linear Start Over playback, which allows the viewer to start over from the beginning of a programme, while using ***Trick Play*** during the playing of previously aired content. This form of playback differs from Linear Start Over, where a broadcast break is replaced by content that matches the duration of the underlying break, in that the break can be expanded or contracted in duration as required.
Non-Linear	This may refer to ***Video On Demand*** or ***NLSO*** or ***Live Pause***, allowing for the time-shifted viewing of Live streams.
Non-Linear Creatives	The Non-Linear creatives associated with an ad are typically static images that are presented in an overlay on top of the video content. In some cases (such as for VAST 2.0 style Non-Linears, there is no defined location and it is left up to the implementer to decide how the creatives should be displayed.
Prefetch	Prefetching involves calling ads for the **next** ad break (as opposed to the current break), giving ad decisioning systems a longer window in which to respond. Prefetching lessens the load on ad servers during those moments when all viewers go to an ad break simultaneously, providing fast and consistent response times when entering an ad break.
Scrub Control	A form of ***Trick Play*** that allows you to use a control bar to ***Seek to*** a position that is either in front of, or behind, the current playhead position. If the playback policy allows, the playhead position will follow.
Seek to	When using ***Non-Linear*** playback, a `canSeek` method can be invoked (subject to the terms of the playback policy) to allow the player to 'seek to' move the playhead to a new position.

Copyright © 2009-2018 Yospace Technologies Ltd.

Trick Play	A feature of digital video systems that mimics the interaction via fast-forward and rewind operations that was previously provided by analogue systems such as VCRs. Refer also to ***Scrub Control***.
Video Ad Serving Template (VAST)	This refers to a specification defining a universal XML schema for serving ads to digital video players. It describes expected video player behaviour when executing VAST--formatted ad responses. For more details on the current version supported by Yospace, please tefer to **https://www.iab.com/guidelines/digital-video-ad-serving-template-vast-3-0/** .
Video Multiple Ad Playlist (VMAP)	A specification for an XML template that describes the structure for ad inventory insertion. A VMAP document defines the ad breaks within video content; the number and placement breaks as well as the adverts within each break. The VMAP document contains a VAST payload block for each ad break that it defines. For more details, please refer to **https://www.iab.com/guidelines/digital-video-multiple-ad-playlist-vmap-1-0-1/**.
Video On Demand (VOD)	A ***Non-Linear*** means of playback that allows the viewer to watch video content at a time of their choosing, rather than at the specified broadcast time.
Video Player Ad Interface Definition (VPAID)	A specification that defines a comnmon interface between the video player and the ad unit, and provides the means for user interactivity. For more details on VPAID compliance, please refer to **https://www.iab.com/?s=vpaid**.
yospaceCDS (CDS)	This constitutes the entire Yospace platform, being a management system required for the operation and delivery of Yospace components within a network. The CDS provides persistence, configuration, asset management, ingestion, encoding and request access functions for other Yospace components within the network.

 Copyright © 2009-2018 Yospace Technologies Ltd.