iab.

VIDEO PLAYER-AD INTERFACE DEFINITION (VPAID)

VERSION 2.0

Released April 10th, 2012

This document was developed by the IAB Digital Video Committee

The Video Player-Ad Interface Definition (VPAID) Guidelines was updated to version 2.0 by a working group of volunteers from 42 IAB member companies.

The Working Group was led by Co-Chairs:

- Teg Grenager, Adap.tv
- Payam Shodjai, Google/YouTube

The following IAB member companies contributed to this document:

24/7 Real Media
Adap.tv
Adobe Systems
AppNexus
Auditude
BlackArrow
Brightcove
Brightroll
Cable Television Laboratories, Inc.
CBS Interactive
Comcast Interactive Media
Digital Broadcasting Group
Evidon
FreeWheel
Google & YouTube
HealthiNation
ImServices Group Ltd.
Innovid, Inc.
Kantar Video
LiveRail, Inc.
MediaMind
Microsoft
Mixpo
NBC Universal Digital Media
New York Times
Nielsen
OneScreen
Ooyala
PointRoll
SeaChange International
SpotXchange
TargetSpot
Tremor Video
TRUSTe
TubeMogul
Turner Broadcasting System
Unicast
Vindico
Weather.com
Yahoo!
YuMe

The IAB leads on this initiative were Chris Mejia and Katie Stroud

Contact adtechnology@iab.net to comment on this document. Please be sure to include the version number of this document (found on the bottom right corner on this page).

ABOUT THE IAB'S DIGITAL VIDEO COMMITTEE

The Digital Video Committee of the IAB is comprised of over 180 member companies actively engaged in the creation and execution of digital video advertising. One of the goals of the committee is to implement a comprehensive set of guidelines, measurement, and creative options for interactive video advertising. The Committee works to educate marketers and agencies on the strength of digital video as a marketing vehicle. A full list of Committee member companies can be found at: www.iab.net/digital_video_committee

This document is on the IAB website at: http://www.iab.net/videosuite/vpaid

Table of Contents

Executive Summary

The IAB's Video Player Ad-Serving Interface Definition (VPAID) establishes a common interface between video players and ad units, enabling a rich interactive in-stream ad experience.

In-stream video advertisers have two important execution goals for the delivery of their video ad campaigns: a) provide viewers a rich ad experience, and b) capture ad playback and user-interaction details that report on the viewed ad experience. To achieve these goals in a world without common video player functionality, advertisers would have to develop multiple specialized versions of their ad creative for every unique video player—an expensive proposition that doesn't scale well.

The Video Ad-Serving Template (VAST), another IAB specification, provides a common ad response format for video players that enables video ads to be served across all compliant video players. However, VAST alone does not provide support for rich interactivity. VAST alone only supports relatively simple in-stream video ad formats that are not executable. These simple ad formats do not provide an interactive user experience, and do not allow the advertiser to collect rich interaction details.

Layering VPAID onto VAST offers an enhanced solution. VPAID establishes a common communication protocol between video players and ad units that allows a single "executable ad" (one that requires software logic to be executed as part of ad playback) to be displayed in-stream with the publisher's video content, in any compliant video player. Furthermore, it enables the executable ad unit to expect and rely upon a common set of functionality from the video player. VPAID enables the video player to expect and rely upon a common set of functionality from the executable ad unit. The significance is that advertisers using VPAID ads can provide rich ad experiences for viewers and collect ad playback and interaction details that are just as rich as the ad experience.

With the adoption of VPAID, advertisers have more control over the display experience in their video campaigns. Also, as VPAID compliant video players enable a more diverse and interactive set of video advertising, VPAID compliant publishers should expect to sell more in-stream video inventory.

With VPAID, the IAB aims to address the following market inefficiencies for publishers, advertisers, and vendors by:

- Increasing common video ad supply technology so that video publishers can readily accept video ad serving from agency ad servers and networks;
- Providing common technology specifications for advertisers to develop against, thereby decreasing the cost of creative production and thus increasing business ROI;
- Improving video ad supply liquidity, thus decreasing the cost of integration with each publisher.

To improve the interactive ad experience in video players, publishers should build their video players to the VPAID specifications outlined in this document. These specifications were defined with creativity and innovation in mind and should not limit video player design.

As with all IAB guidelines and specifications, this document will be updated as video advertising progresses and new ad formats become more widely adopted.

Intended Audience

Anyone involved in the in-stream (also referred to as "in-player") video advertising supply chain can benefit from being familiar with this specification, however implementation details are targeted toward the developers of executable in-stream video ads, and video player software developers. Specifically, video software engineers and video product managers should use this document as a guide when implementing technology designed to support VPAID.

Other IAB Digital Video Guidelines

The VPAID solution is part of a larger initiative to improve the liquidity of digital video advertising. This initiative includes the following:

- *Video Ad Serving Template (VAST)*
- *Video Ads Multi-Playlist (VMAP)*
- *Digital Video Measurement Guidelines*
- *Digital Video Ad Format Guidelines and Best Practices*
- *Digital Video In-Stream Ad Metrics Definitions*

Updates in VPAID Version 2.0

VPAID 1.0 enabled cross-platform support for rich in-stream video ads. As VPAID acceptance has begun to permeate the industry, VPAID 2.0 brings enhancements and additions that provide support for more interactive capabilities and improved reporting.

Updates in VPAID 2.0 are summarized below:

- **Document Rewrite:** The content in VPAID 2.0 has been reorganized and simplified where possible to improve the flow of explanations, while also empowering non-technical readers to understand VPAID.
- **VPAID and VAST:** A valid VPAID object can be used in conjunction with the IAB Video Ad-Serving Template (VAST) and is highly recommended, as VAST ads that include VPAID protocols can play in both VAST and VPAID-enabled video players. VPAID 2.0 includes details about how to use VPAID protocols in a VAST ad unit.
- **Support for HTML 5:** HTML 5 is an emerging Web syntax that has the potential to enable cross-platform/cross-device support for the latest trends in multimedia. Details for HTML 5 use of VPAID are included in this update. See Section 8 for details.
- **Technical feature updates:** In order to support added features for advanced display and reporting, the following properties, methods and dispatched events have been added or changed in this update:

  Methods

  - **resizeAd()**: clarification has been added about how to use this method in fullscreen mode

- **skipAd()**: added to enable the video player to include controls for allowing its audience to skip ads. The new `AdSkipped` event is dispatched by the ad unit in response to this call.

Properties

- **adLinear**: clarification added to indicate when the property should be used
- **adWidth**: added to provide current width of ad unit after ad has resized
- **adHeight**: added to provide current height of ad unit after ad has resized
- **adDuration**: reports total duration to more clearly report on the changing duration, which is confusing when both remaining time and duration can change
- **adCompanions**: included to support ad companions in VPAID, when companion information is not available until after the VPAID .swf file has already loaded.
- **adIcons**: included to support various industry programs which require the overlay of icons on the ad.
- **adSkippableState**: in support of skippable ads, this feature enables the video player to identify when the ad is in a state where it can be skipped.

Dispatched Events

- **AdStopped**: clarification added to indicate that the `AdStopped` event is to be used as a response to `stopAd()` (or dispatched when the ad has stopped itself) rather than as a request to the video player to call `stopAd()`.
- **AdSizeChange**: added to enable confirmation to a `resizeAd()` method call from the video player
- **AdDurationChange**: instead of reporting `AdRemainingTimeChange`, `AdDurationChange` reports changes on the total duration that can change with user interaction. In the event of an `AdDurationChange`, both `adRemainingTime` and `adDuration` properties are updated
- **AdInteraction**: added to capture users' interactions with the ad other than the `ClickThru` events.
- **AdSkipped**: added to support ads that include skip controls. This event can be triggered by controls in the ad unit or in response to the video player calling the `skipAd()` method.
- **AdSkippableStateChange**: added to support skippable ads, this event enables the ad unit to report when the ad is in a skippable state

1 Introduction

Ads served into a publisher's video content stream render within the video player environment inline with the video content, while traditional banners render on the webpage itself. The video player makes the ad call (instead of the browser) and interprets the ad response from the ad server. The video player is also the run-time environment for the in-stream video ad unit and controls its lifecycle.

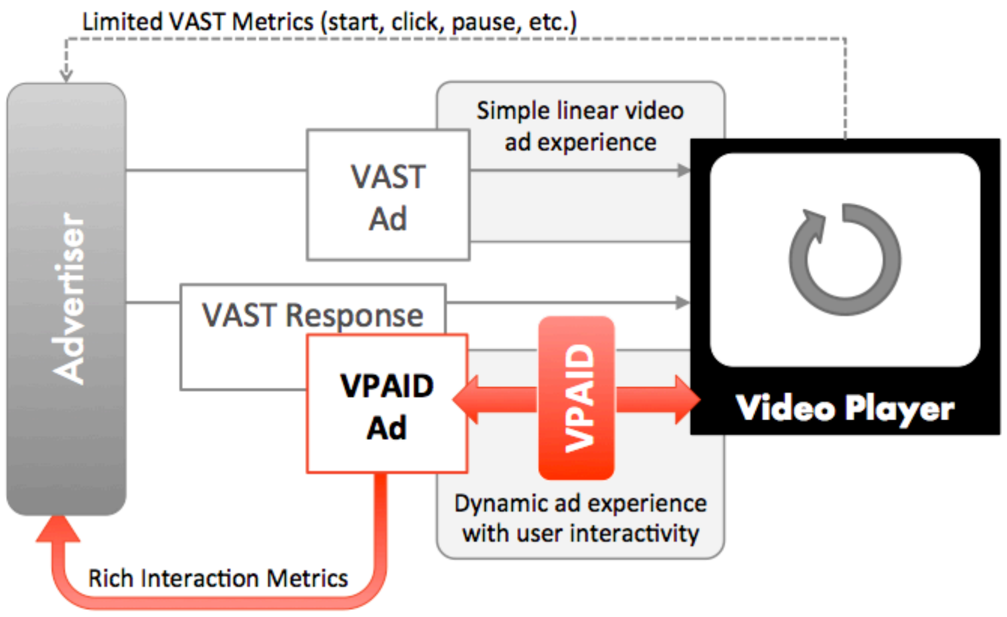
This relationship between the video player and the in-stream video ad unit becomes more complex as technology advances. Many ads displayed in-stream with video content are executable: they contain logic which must be executed by the video player in order to provide an interactive user experience and/or to support the collection of a rich set of interaction metrics. If video players are to support these executable ad units, they require an API to support ongoing communication between the players and the ad units.

VPAID offers an API to facilitate increasing complexity in video advertising, enabling video players to accept more ads, and a platform that offers more value to advertisers and publishers alike.

1.1 VPAID Enables Ad Interactivity

Relative to standard linear video ads, executable ads provide a far more dynamic advertising experience for consumers, because they enable rich user interaction. For example, an executable in-stream video ad unit may include additional clickable links and buttons that allow the user to view a longer form of the ad, share the ad using social media, or request more information, such as store locations or show times. These rich interactions can be captured in real-time metrics, offering advertisers interaction reports that are just as rich as the advertising experience. The VPAID API allows advertisers to distribute these executable ads across a broad set of compliant video players and publishers, to many compliant publishers.

In the diagram below, both a VPAID ad unit and a VAST ad unit have been served to a video player:


Limited VAST Metrics (start, click, pause, etc.)
Advertiser
Simple linear video ad experience
VAST Ad
VAST Response
VPAID Ad
VPAID
Video Player
Dynamic ad experience with user interactivity
Rich Interaction Metrics

The VAST ad unit is served as a package: everything needed to serve the ad is included in the VAST XML document for the ad unit. The video player can only report on certain essential events like start, click, pause, play, etc. User interactions can't change the ad playback in any way. The ad plays for a set duration with only basic interactions.

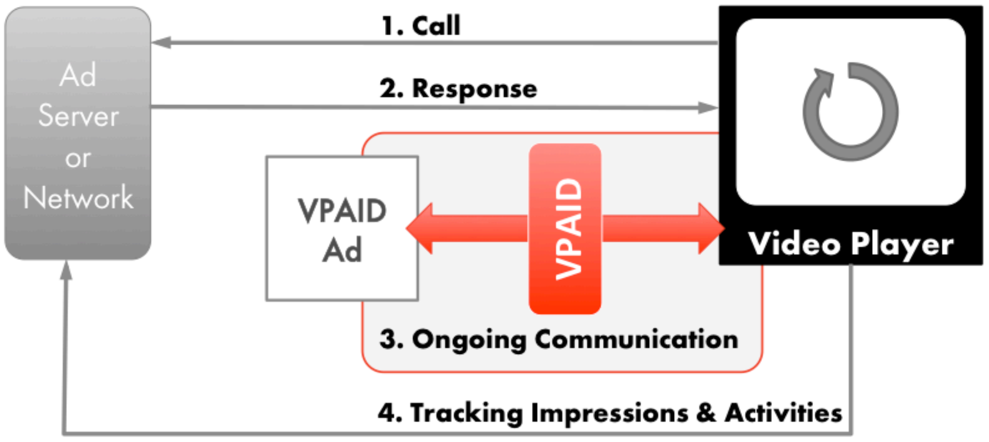
With a VPAID-compliant executable ad unit, the interactive experience engages users and offers a dynamic brand opportunity for the advertiser. Such interactions happen in real time and can be reported to the advertiser.

Because VPAID offers such a dynamic advertising experience and a growing acceptance in the industry, advertisers are increasing their use of VPAID-enabled ad units. Publishers offering video players that are VPAID-enabled can accept more of these ads—often for premium compensation.

1.2 Video Ad Flow with VPAID

VPAID opens a line of communication between an executable ad unit and a video player. This open line of communication makes it possible for the two to interact. For example, if an ad is designed to play at a particular moment during a video, the video player can notify the ad unit when the moment is reached so the ad can display as designed. Likewise, if the state of an ad changes because of user interaction, notice of the change from the video player can trigger the ad unit to react.

The diagram below displays a simplified example of how a video ad unit and video player interact using VPAID:



1. **Call:** The player makes an ad call to the ad server. The format for this ad call is not specified in these guidelines in order to maintain flexibility for the many variations of ad server formats.
2. **Response:** The ad server responds with a VAST XML containing a VPAID-compliant executable ad unit.
3. **Ongoing Communication:** The video player and the ad unit remain in communication as the ad executes and displays to the user. Using VPAID, the video

player can get and set properties for the ad unit, and the ad unit can dispatch events to the video player.

4. **Tracking Impressions & Activities:** The video player and the ad unit can each send impression and activity tracking requests to their respective ad servers (ad server tracking is not specified by VPAID).

1.3 Cross-Platform Support

VPAID is designed as a platform-independent API. The API must be defined individually for each platform that the video player and in-line video ad unit run on: AS2, AS3, Silverlight, JavaScript. However, to be clear, it does not help bridge platforms. Nothing in VPAID enables an executable ad unit designed to run on one platform such as Flash™, run in a video player that does not support that platform (i.e. built using JavaScript).

1.4 Scope and Limitation

VPAID is designed for video players that are capable of loading and running executable in-stream video ad units using one of the supported platforms. The scope of this guideline is to provide recommendations that enable a common interface between an in-stream video ad unit and video player so that the video player can execute on an ad unit's interactive instructions and record/report the results. Other uses are possible, but certain limitations and specific scope should be noted.

Note: VPAID is not limited to an interface specifically between an ad unit and a video player. Other application interface layers may work with VPAID as long as, when combined, the VPAID specification and interface are respected.

The following technical scenarios are out of scope:

Bridging Video Player and Ad Unit Technologies: While VPAID establishes a common interface to bridge communication between an ad unit and the video player, the technology supports ad unit display behavior and tracking capabilities. Specifications for playing an ad unit of one technology within the video player of another technology are not included. For example, VPAID doesn't enable a Flash™ player to play Silverlight™ ads. Mechanisms exist for addressing the incompatibility, but are not covered as a feature of VPAID at this time.

Preloading Logic: VPAID enables the video player to preload an ad unit before rendering it. However, these guidelines don't address the logic for how to preload an ad or how to deal with the ramifications that preloading has on impression counts.

Impression and Display Tracking Guidelines: VPAID enables rich display and interaction tracking and provides details about how this works. However, the timing and method for what and when to count is out of scope for this document. Please refer to the IAB *Digital Video Ad Measurement Guidelines* for impression and display tracking guidelines.

Ad Rule Management: The logic for managing ad rules and any included business rules for deciding when to make an ad call and which type of ad to call are out of scope for this document.

1.5 VPAID, MRAID and Mobile Technology

VPAID was designed to work across multiple platforms, many of which include various mobile devices. However, VPAID interactions are limited where specific mobile interactions apply. For example, VPAID isn't designed to track use of the accelerometer in mobile devices such as when the user "shakes" the mobile device to play a song or initiate some other action.

IAB offers a specification for managing interactions between mobile apps and rich media ad units. The Mobile Rich Media Ad Interface Definition (MRAID) can be used for ads targeted specifically to mobile devices where ads are played in mobile applications, but VPAID is for use in campaigns targeted to digital video across any technical video player platform, including mobile.

2 API Reference

This section provides general information about VPAID such as the guiding principles for design, video player and ad unit requirements, dependency on XML, and using VPAID in a VAST context.

2.1 Guiding Principles

VPAID was designed using the following guiding principles:

1. **Generalized:** the API needs to be generalized enough to support emerging interactive ad formats. For example, the API should enable video players to position ads in time and space when sufficient data is available.
2. **Simple:** the API needs to be as simple as possible while still satisfying the ability to offer generalized support for emerging ad formats (as stated in principle 1 above).
3. **Shift the API burden on to the video player:** The video player must implement the API so that it can display any VPAID ads without requiring API implementation in simple ad formats.
4. **Video Player in control:** the API is designed in a way that the ad unit could request certain services from the video player, but the video player ultimately remains in control of the overall run-time environment. For example, the video player has the ability to unload a misbehaving ad unit.
5. **Keep the ad unit portable:** API specifications should allow the ad unit to be independent of the ad request parameters, XML schema used to traffic the ad unit, reporting beacon formats, etc. Separating the ad unit from specific technologies enables the ad unit to be portable across the multiple ad platforms.
6. **Consistency across technology implementations:** the API should function across all ad development solutions such as: AS2, AS3, Silverlight, JavaScript, etc.

7. **Compatibility with previous versions:** In VPAID 2.0 changes and additions were added with respect to compatibility with previous versions of VPAID. A VPAID 2.0 video player should operate without error when interfacing with an ad unit that implements an earlier version of VPAID such as VPAID 1.1. Likewise, a VPAID 2.0 ad unit should display properly in a VPAID 1.1 video player; however, version 2.0 functionality would not be supported. Also, both the ad unit and the video player may need to relax version control in older versions for backwards compatibility. Please report compatibility issues to adtechnology@iab.net, noting the versions of both ad and player tested.

2.2 Notation

The following table describes notation conventions used in VPAID.

`name : Type`	Specifies a variable, property or method name followed by data types or return data type.
`{}`	Specifies an object with `name : Type` properties
`Array<Type>`	Specifies an array of a particular data type.
`[]`	Specifies optional parameters or data structure properties.
|	Specifies "or" for data structure properties.
*	Specifies repeat 0 or more times
+	Specifies repeat 1 or more times.

2.3 Video Player Requirements

The video player must implement ad loading, check for presence of `VPAID`, and if present, implement the correct VPAID version. Recovery mechanisms should also be in place should the ad unit fail to follow specified protocols correctly. For example, if the video player does not receive the `AdStopped` event from an ad unit after sending a call to `stopAd()`, the video player should be prepared to react appropriately.

Without recovery mechanisms, an improperly scripted ad unit may cause the video player to crash. Uncaught exceptions should never be thrown while handling VPAID events from the ad unit. See Implementation sections for more specific requirements. Section 3.4 provides details on error handling and recommended recovery action for specific timeout situations.

2.3.1 Displaying the Ad Unit Creative

The video advertising industry expects that an impression has indicated that an ad was viewed by an end user. Therefore, the video player should ensure that no other visual elements from the video player, the webpage, or other ads, display over a video ad that is currently in progress.

Also, since video ad creative are expected to adhere to industry guidelines that include a "Close X" button, the video player should not include its own close button for VPAID ad units.

2.4 Ad Unit Requirements

If the ad unit implements VPAID, it must indicate the correct version. The ad unit must implement all methods and properties listed, but can either decline from responding or return a value that indicates that the method or property is not supported. For example, if the ad unit implements VPAID 2.0, then values for properties such as `adCompanions` or `adDuration` should be provided but can provide values that indicate that the property is not in use.

The ad unit should never allow uncaught exceptions to be thrown during calls into VPAID from the video player. See Implementation sections for more specific requirements.

2.5 Using VPAID in Conjunction with VAST

Where possible, VPAID should be sent to the video player in a VAST ad response. This section provides information about how VAST and VPAID work together.

2.5.1 API Framework

When a VPAID ad unit is referenced from a VAST document, the value for the `apiFramework` attribute in the `<MediaFile>` element must be VPAID (all caps). This attribute identifies the VPAID API for the creative. Version information should be handled by the VPAID `handshakeVersion()` call (rather than identified in the VAST file). See Section 3.1.1 for more information.

2.5.2 How to Initialize the VPAID Ad Unit

When using VAST, the only way to pass information from VAST into a VPAID ad unit is using the `<AdParameters>` elements in VAST.

Linear and nonlinear VPAID ad units are loaded in a VAST context using the VPAID `initAd()` method. Depending on the ad unit's linearity, the `creativeData` parameter in this method must be set to accept `<AdParameters>` value from either the `<Linear>` or the `<NonLinear>` element from the VAST document. In VAST, the `AdParameters` value may be wrapped in a CDATA block but the ad unit may have trouble processing the`]>` tring used to close a CDATA block.

To identify the platform used to implement the ad unit (i.e. MIME types such as application/x-shockwave-flash or application/javascript), use the `type` attribute on the `<MediaFile>` element in VAST if it's a linear creative. For a nonlinear creative, if using `StaticResource` then use the `creativeType` attribute, if using HTMLResource, the implication is that the creative is javascript. If two or more technology platforms are provided through multiple media files, the video player should use the one closest to its own technology.

In the `initAd()` function call, the `environmentVars` function parameter is optional. It should be used when certain information is unavailable as the VPAID ad unit is requested, but made known as the VPAID ad unit loads.

2.5.3 How to handle event tracking

When a VPAID ad unit is sent to the video player in a VAST response, the video player may receive VPAID events from the ad unit. In this situation, the video player should send requests to the tracking URI(s) in the VAST response for the VAST event that corresponds to the VPAID event.

For example, if the video player received the VPAID event `AdVideoStart`, and there was a tracking URI in the VAST response for the corresponding VAST event `"start"`, then the video player should send a request to VAST tracking URI. Note that some VPAID events do not have corresponding VAST events, such as the `AdLinearChange` event.

Duplicate events may be recorded if the VPAID ad unit sends tracking requests to the event-tracking URIs while also dispatching VPAID events to the video player because the video player may also send tracking requests to the same event URIs.

Table 2.5.3 below shows the correspondence between VPAID events and VAST events. When served in a VAST response, VPAID events in the left column of the table should trigger the corresponding VAST event (if any) in the column on the right.

Receive VPAID event…	**…triggers VAST TrackingEvent**
AdLoaded	-
AdSkipped	skip
AdStarted	creativeView
AdStopped	-
AdLinearChange	-
AdExpandedChange	-
AdRemainingTimeChange	-

AdVolumeChange	if (currentVolume==0 and lastVolume>0) then mute
	if (currentVolume>0 and lastVolume==0) then unmute

AdImpression	<Impression> (the VAST element; not an event type)
AdVideoStart, AdVideoFirstQuartile, AdVideoMidpoint, AdVideoThirdQuartile, AdVideoComplete	start, firstQuartile, midpoint, thirdQuartile, complete
AdClickThru	<ClickTracking> (ClickTracking is a VAST element under <VideoClicks>, not an event type)

Receive VPAID event…	…triggers VAST TrackingEvent
AdInteraction	-
AdUserAcceptInvitation, AdUserMinimize, AdUserClose	acceptInvitation, collapse, close
AdPaused, AdPlaying	pause, resume
AdLog	-
AdError	error (with error code 901, as noted in VAST 3.0)

Table 2.6.3: VPAID/VAST Tracking Event Map

2.5.4 How to handle VPAID clicks in VAST context

The following logic determines how the video player should respond to clicks:

- The `event.data.url` is optional
- If `event.data.playerHandles` is true and e.data.url is:
    a. **Not defined**: then the video player must use the VAST element, `VideoClicks/ClickThrough`.
    b. **Defined**: then the video player must use `event.data.url`.
- If `event.data.playerHandles` is false, then the video player doesn't open the landing page URL. The ad unit is responsible for opening the landing page URL in a new window in this case.

Note: Regardless of the state of the `event.data.handles`, the video player must request the resource specified by the URIs in the `<VideoClicks>` and `<ClickTracking>` elements of VAST when it receives an `AdClickThru` VPAID event.

2.5.5 How to Interpret the VAST Linear/Duration Element

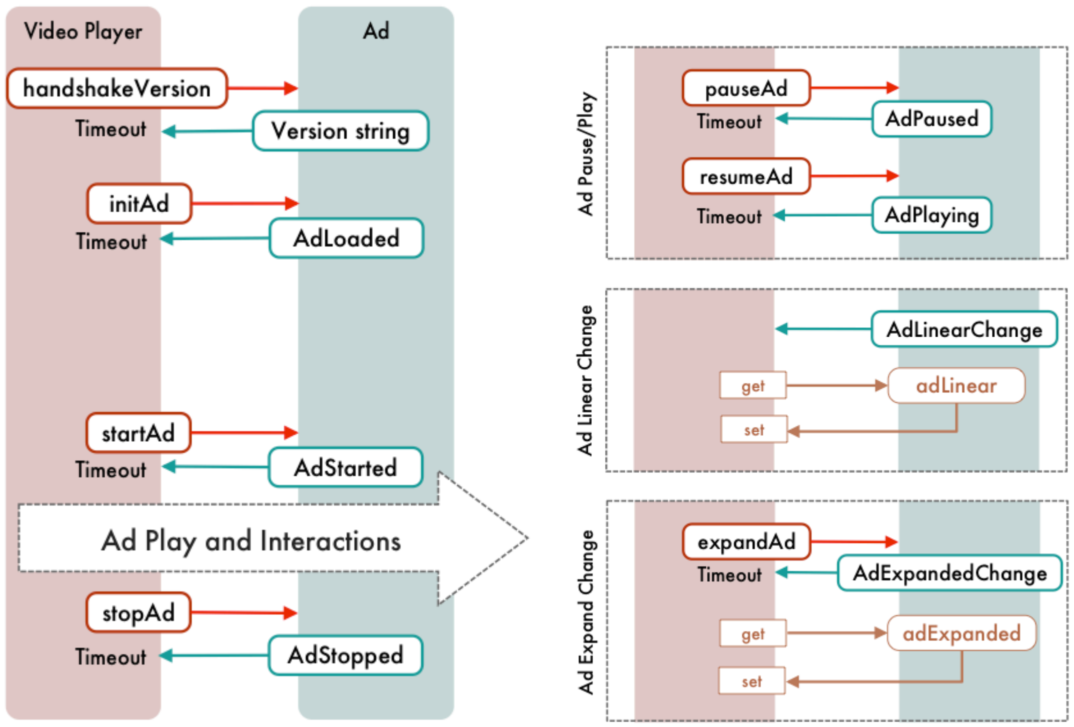
VAST doesn't support interactive ad properties like the variable duration supported in VPAID ad units that can change in response to user interaction. When a VPAID ad unit has variable duration, the `Linear/Duration` element in VAST should be identified as the duration of the interactive ad unit before any user interaction.

- In VPAID 1.0, the value of `Linear/Duration` element in VAST should use the value of the VPAID property, `adRemainingTime`, collected at the time of the `AdStarted` event (before user interaction).
- In VPAID 2.0, use the VPAID property, `adDuration`, instead.

3 VPAID Protocol Details

VPAID protocols enable the video player and the ad unit to communicate the state of the ad playback and include methods, properties and events.

The diagram below provides an example of how methods and events are used in sequence by the ad unit and video player. The video player on the left of the diagrams makes calls (red) to the ad unit on the right. The ad unit responds by dispatching events (teal) to the video player. Sometimes, the video player checks (or "gets") ad unit properties (yellow) and uses the information to modify its UI or manage content video playback.



Whenever the video player must wait for the ad unit to respond, the ad unit may take too long to send the appropriate event. In each of these cases, the video player should implement timeout instructions for how to respond in the absence of an expected ad unit response. Please see 0

Error Handling and Timeouts for more information on timeouts.

The following sections provide descriptions for all VPAID protocols.

3.1 Methods

All methods are called by the video player on the ad unit's VPAID member property object.

The video player must refrain from calling any methods (besides `handshakeVersion()` or `initAd()`) on the ad unit or access any properties until after the `AdLoaded` event has been dispatched. The ad unit may not be able to provide any information or response until after the ad unit has loaded and `AdLoaded` has been dispatched.

3.1.1 handshakeVersion

handshakeVersion(playerVPAIDVersion : String) : String

The video player calls handshakeVersion immediately after loading the ad unit to indicate to the ad unit that VPAID will be used. The video player passes in its latest VPAID version string. The ad unit returns a version string minimally set to "1.0", and of the form "major.minor.patch" (i.e. "2.1.05"). The video player must verify that it supports the particular version of VPAID or cancel the ad.

In VPAID 2.0, updates were made with support of version 1.0 and later in mind. Video players of version 2.0 should correctly display ads of version 1.0 and later, and video players of 1.0 and later should be able to display ads of version 2.0, but not all features will be supported. Testing should included ad play in different version environments to verify any compatibility issues.

Static interface definition implementations may require an external agreement for version matching, in which case the handshakeVersion method call isn't necessary. However, when dynamic languages are used, the ad unit or the video player can adapt to match the other's version if necessary. Dynamic implementations may use the handshakeVersion method call to determine if an ad unit supports VPAID. A good practice is to always call handshakeVersion even if the version has been coordinated externally.

3.1.2 initAd()

initAd(width : Number, height : Number, viewMode : String, desiredBitrate : Number, [creativeData : String], [environmentVars : String]) : void

After the ad unit is loaded and the video player calls handshakeVersion, the video player calls `initAd()` to initialize the ad experience. The video player may preload the ad unit and delay calling `initAd()` until nearing the ad playback time; however, the ad unit does not load its assets until `initAd()` is called. Once the ad unit's assets are loaded, the ad unit sends the `AdLoaded` event to notify the video player that it is ready for display. Receiving the AdLoaded response indicates that the ad unit has verified that all files are ready to execute.

Parameters used in the `initAd()` method:

- **width:** indicates the available ad display area width in pixels
- **height:** indicates the available ad display area height in pixels
- **viewMode:** indicates either “normal”, “thumbnail”, or “fullscreen” as the view mode for the video player as defined by the publisher. Default is “normal”.
- **desiredBitrate:** indicates the desired bitrate as number for kilobits per second (kbps). The ad unit may use this information to select appropriate bitrate for any streaming content.
- **creativeData:** (optional) used for additional initialization data. In a VAST context, the ad unit should pass the value for either the Linear or Nonlinear AdParameter element specified in the VAST document.
- **environmentVars:** (optional) used for passing implementation-specific runtime variables. Refer to the language specific API description for more details.

3.1.3 resizeAd()

resizeAd(width : Number, height : Number, viewMode : String) : void

The `resizeAd()` method is only called when the video player changes the width and height of the video content container, which prompts the ad unit to scale or reposition. The ad unit then resizes itself to a width and height that is equal to or less than the width and height supplied by the video player. Once resized, the ad unit writes updated dimensions to the `adWidth` and `adHeight` properties and sends the `AdSizeChange` event to confirm that it has resized itself.

Calling resizeAd() is solely for prompting the ad to scale or reposition. Use expandAd() to prompt the ad unit to extend additional creative space.

The parameters for this method call are:

- **Width/Height:** The maximum display area allotted for the ad. The ad unit must resize itself to a width and height that is within the values provided. The video player must always provide width and height unless it is in fullscreen mode. In fullscreen mode, the ad unit can ignore width/height parameters and resize to any dimension.
- **ViewMode:** Can be one of “normal” “thumbnail” or “fullscreen” to indicate the mode to which the video player is resizing. Width and height are not required when viewmode is fullscreen.

By using the `resizeAd()` method to resize, the video player enables the ad unit to resize itself and report its dimensions to the video player. This method is preferred over the video player using its own technology to set the ad size upon video player resize. In fact, the video player should never set the width and height properties of the ad unit. Instead, the video player can get the `adWidth` and `adHeight` properties to verify that the ad unit has resized itself to within the supplied dimensions.

3.1.4 startAd()

startAd() : void

`startAd()` is called by the video player when the video player is ready for the ad to display. The ad unit responds by sending an `AdStarted` event that notifies the video player when the ad unit has started playing. Once started, the video player cannot restart the ad unit by calling `startAd()` and `stopAd()` multiple times.

3.1.5 stopAd()

stopAd() : void

The video player calls `stopAd()` when it will no longer display the ad or needs to cancel the ad unit. The ad unit responds by closing the ad, cleaning up its resources and then sending the `AdStopped` event. The process for stopping an ad may take time. Please see section 0

Error Handling and Timeouts for more information on error reporting and timeouts.

3.1.6 pauseAd()

pauseAd() : void

The video player calls `pauseAd()` to prompt the ad unit to pause ad display. The ad unit responds by suspending any audio, animation or video and then sending the `AdPaused` event. Instead of simply stopping animation and perhaps dimming display brightness, the ad unit may choose to remove UI elements. Once `AdPaused` is sent, the video player may hide the ad by adjusting the visibility setting for the display container. If the video player does not receive the `AdPaused` event after a `pauseAd()` call, then either the ad unit cannot be paused or it failed to send the `AdPaused` event. In either case, the video player should treat the lack of response as a failed attempt to pause the ad.

3.1.7 resumeAd()

resumeAd() : void

Following a call to `pauseAd()`, the video player calls `resumeAd()` to continue ad playback. The ad unit responds by resuming playback and sending the `AdPlaying` event to confirm. If the video player does not receive the `AdPlaying` event after a `resumeAd()` call, then either the ad unit cannot resume play or it failed to send the `AdPlaying` event. In either case, the video player should treat the lack of response as a failed attempt to initiate resumed playback of the ad.

3.1.8 expandAd()

expandAd() : void

The video player calls `expandAd()` when the timing is appropriate for an expandable ad unit to play at additional interactive ad space, such as an expanding panel. The video player may use this call when it provides an "Expand" button that calls `expandAd()` when clicked. The ad unit responds by setting the `adExpanded` property to true and dispatching the `AdExpandedChange` event, to confirm that the `expandAd()` call caused a change in behavior or appearance of the ad.

3.1.9 collapseAd()

collapseAd() : void

When the ad unit is in an expanded state, the video player may call `collapseAd()` to prompt the ad unit to retract any extended ad space. The ad unit responds by setting the `adExpanded` property to false and dispatching the `AdExpandedChange` event, to confirm that the `collapseAd()` call caused a change in behavior or appearance of the ad.

The video player can verify that the ad unit is in an expanded state by checking the value of the `adExpanded` property at any time. The ad unit responds by restoring ad dimensions to its smallest width and height settings and setting its `adExpanded` property to "false."

The expectation is that the smallest UI size should have the least visible impact on the user, for best user-experience. Therefore, if the ad unit has multiple collapsed states, such as a minimized "pill" and a larger click to video banner state (see `expandAd()` for more details), then the `collapseAd()` call should result in the minimized "pill" state. Ad designers should condider implementing both collapsed states in all of their video ads for best user-experience. However, only one collapsed state is required.

Note: If the video player does not call `collapseAd()`, and the ad unit instead initiates a collapse on its own by setting `adExpanded` to false and sending the `AdExpandedChange` event, then the ad is free to choose any collapsed state it supports and not necessarily the smallest UI size.

3.1.10 skipAd()

skipAd() : void

The `skipAd()` method is new in VPAID 2.0.

This method supports skip controls that the video player may implement. The video player calls `skipAd()` when a user activates a skip control implemented by the video player. When called, the ad unit responds by closing the ad, cleaning up its resources and sending the `AdSkipped` event.

The player should check the ad property '`adSkippableState`' before calling `adSkip()`. `adSkip()` will only work if this property is set to true. If player calls `adSkip()` when the '`adSkippableState`' property is set to false, the ad can ignore the skip request.

The process for stopping an ad may take time. Please see section 3.4 Error Handling and Timeouts for more information on error reporting and timeouts.

An `AdSkipped` event can also be sent as a result of a skip control in the ad unit and the video player should handle it the same way it handles an `AdStopped` event. If a skip control in the ad unit triggers the `AdSkipped` event, the video player may also send an `AdStopped` event to support video players using an earlier version of VPAID. The `AdStopped` event sent right after an `AdSkipped` event can be ignored in video players using VPAID 2.0 or later.

Also, if the VPAID version for the ad unit precedes version 2.0, the ad unit will not acknowledge a `skipAd()` method call. Skip controls in the video player should use the `stopAd()` method to close skipped ads that use earlier versions of VPAID.

3.2 Properties

The video player can access all properties on the ad unit's VPAID member property object. If the property is a get property, the ad unit writes to the property and the video player reads from the property. If the property is set property, the video player writes to the property and the ad unit reads from the property.

3.2.1 adLinear

`get adLinear : Boolean`

The `adLinear` Boolean indicates whether the ad unit is in a linear (true) or non-linear (false) mode of operation. The `adLinear` property should only be accessed after the ad unit has dispatched the `AdLoaded` event or after an `AdLinearChange` event.

The `adLinear` property affects the state of video content. When set to true, the video player pauses video content. If set to true initially and the ad unit is designated as a pre-roll (defined externally), the video player may choose to delay loading video content until ad playback is nearly complete.

3.2.2 adWidth

`get adWidth : Number`

The `adWidth` property is new to VPAID 2.0.

The `adWidth` property provides the ad's width in pixels and is updated along with the `adHeight` property anytime the `AdSizeChange` event is sent to the video player, usually after the video player calls `resizeAd()`. The ad unit may change its size to width and height values equal to or less than the values provided by the video player in the `Width` and `Height` parameters of the `resizeAd()` method. If the `ViewMode` parameter in the `resizeAd()` call is set to "fullscreen," then the ad unit can ignore the Width and Height values of the video player and resize to any dimension. The video player may use `adWidth` and `adHeight` values to verify that the ad is appropriately sized.

Note: `adWidth` value may be different than `resizeAd()` values
The value for the `adWidth` property may be different from the width value that the video player supplies when it calls `resizeAd()`. The `resizeAd()` method provides the video player's *maximum allowed* value for width, but the `adWidth` property provides the ad's actual width, which must be equal to or less than the video player's supplied width.

3.2.3 adHeight

`get adHeight : Number`

The `adHeight` property is new to VPAID 2.0.

The `adHeight` property provides the ad's height in pixels and is updated along with the `adWidth` property anytime the `AdSizeChange` event is sent to the video player, usually after the video player calls `resizeAd()`. The ad unit may change its size to width and height values equal to or less than the values provided by the video player in the `Width` and `Height` parameters of the `resizeAd()` method. If the `ViewMode` parameter in the `resizeAd()` call is set to "fullscreen," then the ad unit can ignore the `Width` and `Height` values of the video player and resize to any dimension. The video player may use `adWidth` and `adHeight` values to verify that the ad is appropriately sized.

Note: `adHeight` values may be different than `resizeAd()` values
The value for the `adHeight` property may be different from the height value that the video player supplies when it calls `resizeAd()`. The `resizeAd()` method provides the video player's *maximum allowed* value for height, but the `adHeight` property provides the ad's actual height, which must be equal to or less than the video player's supplied height.

3.2.4 adExpanded

```
get adExpanded : Boolean
```

The `adExpanded` Boolean value indicates whether the ad unit is in a state where additional portions it occupies more UI area than its smallest area. If the ad unit has multiple collapsed states, all collapsed states show `adExpanded` being false. There can only be one expanded state for the creative, which for a non-linear ad is usually the largest possible size for the ad unit and may include a linear mode of operation (though setting `adExpanded` to true does NOT imply that the ad unit is in linear mode).

Specifically, a non-linear ad can support one or more collapsed sizes that allow users to view video content reasonably unimpeded. One example of a larger collapsed state is where a nonlinear overaly typically displays across the lower fifth of the video display area. A secondary, smaller collapsed state, often called a "pill" state, might display as a small overlay button with a visible call-to-action.

The video player can check the `adExpanded` property at any time. Use the `AdExpandedChange` event to indicate that the expanded state has changed. If ad is statically sized `adExpanded` is set to false.

3.2.5 adSkippableState

```
get adSkippableState : boolean
```

The `adSkippableState` property is new to VPAID 2.0.

Common to skippable ads is a timeframe for when they're allowed to be skipped. For example, some ads may only be skipped a few seconds after the ad has started or may not allow the ad to be skipped as it nears the end of playback.

The `adSkippableState` enables advertisers and publishers to align their metrics based on what can and cannot be skipped.

The default value for this property is false. When the ad reaches a point where it can be skipped, the ad unit updates this property to true and sends the `AdSkippableStateChange` event. The video player can check this property at any time, but should always check it when the `AdSkippableStateChange` event is received.

3.2.6 adRemainingTime

```
get adRemainingTime : Number
```

The video player may use the `adRemainingTime` property to update player UI during ad playback, such as displaying a playback counter or other ad duration indicator. The `adRemainingTime` property is provided in seconds and is relative to the total duration value provided in the `adDuration` property.

The video player may check the `adRemainingTime` property at any time, but should always check it when receiving an `AdRemainingTimeChange` (in VPAID 1.0) or `adDurationChange` event (in VPAID 2.0). The ad unit should update this property to be current within one second of actual remain time and can be updated once per second during normal playback or up to four times per second (to maintain optimum performance) so that the video player can keep its UI in synch with actual time remaining.

If the property is not implemented, the ad unit returns a value of -1. A value of -2 is returned when time remaining is unknown. Unknown remaining time usually indicates that a user is actively engaged with the ad.

3.2.7 adDuration

```
get adDuration : Number
```

The `adDuration` property is new to VPAID 2.0.

An ad unit may provide the `adDuration` property to indicate the total duration of the ad, relative to the current state of the ad unit. When user interaction changes the total duration of the ad, the ad unit should update this property and send the `adDurationChange` event. The initial value for `adDuration` is the expected duration before any user interaction.

The video player may check the `adDuration` property at any time, but should always check it when receiving an `adDurationChange` event.

If duration is not implemented, the ad unit returns a -1 value. If the duration is unknown, the ad unit returns a -2. Unknown duration is typical when the user has engaged the ad.

3.2.8 adVolume

```
get adVolume : Number
set adVolume : Number
```

The video player uses the `adVolume` property to either request the current value for ad volume (get) or change the value of the ad unit's volume (set). The `adVolume` value is between 0 and 1 and is linear, where 0 is mute and 1 is maximum volume. The video player is responsible for maintaining mute state and setting the ad volume accordingly. If volume is not implemented as part of the ad unit, -1 is returned as the value for `adVolume` when the video player attempts to get `adVolume`. If set is not implemented, the video player does nothing.

3.2.9 adCompanions

```
get adCompanions : String
```

The `adCompanions` property is new to VPAID 2.0.

Companion banners are ads that display outside the video player area to reinforce the messaging provided in the video ad unit. In some cases, a VPAID ad unit may request ads from other ad servers after `initAd()` has been called, and makes a decision about which ad it will display, which may or may not include ad companions. For example, a client-side yield management SDK may wrap itself in a VPAID ad when a native SDK integration might be cumbersome. In this scenario, the ad server that served the initial VAST response may not know which ad will be displayed, and therefore the VAST response itself does not include ad companions.

VPAID 2.0 enables an ad server to serve a VAST response which has no companions, but which does have a VPAID ad unit that pulls in ad companions dynamically based on the ad-serving situation. The video player can then check the VPAID ad unit for ad companions when the VAST response has none.

The video player is not required to poll this property, and because ad companion information from the VAST response takes precedence over VPAID ad companions, the video player should only access this property when the VAST response is absent of any ad companions.

The value of this property is a String that provides ad companion details in VAST 3.0 format for the `<CompanionAds>` element, and should contain all the media files and details for displaying the ad companions (i.e. the format should be of an `InLine` response and not in `Wrapper` format). Also, the value should only include details within the <CompanionAds> element and not an entire VAST response. If any XML elements are included outside of the `<CompanionAds>` element, they must be ignored, including any `<Impression>` elements that might have been included. However, VAST companion ad `<TrackingEvents>` elements for `<Tracking event="creativeView">` must be respected.

If the video player calls for `adCompanions()`, it must wait until after receiving the VPAID `AdLoaded` event, and any companions provided must not display until after the VPAID `AdImpression` event is received. Delaying companion display until after the `AdImpression` event prevents display of any companion banners in the case where the video ad fails to register an impression.

If this property is used but no Companions are available the property should return an empty string "".

Example of a basic `<AdCompanions>` element from VAST as it may be passed in the VPAID `adCompanions` property:

```
<AdCompanions>
     <Companion>
          <StaticResource type="image/jpg">
               <![CDATA[
               http://AdServer.com/120x60companion.jpg
               ]>
          </StaticResource>
          <TrackingEvents>
               <Tracking event=creativeView>
                    <![CDATA[
                    http://AdServer.com/creativeview.jpg
                    ]>
               </Tracking>
          </TrackingEvents>
     </Companion>
</AdCompanions>
```

3.2.10 adIcons

```
get adIcons : Boolean
```

The adIcon property is new to VPAID 2.0.

Several initiatives in the advertising industry involve using an icon that overlays on top of an ad creative to provide some extended functionality such as to communicate with consumers or otherwise fulfill requirements of a specific initiative. Often this icon and its functionality may be provided by a vendor, and is not necessarily served by the ad server or included in the creative itself.

One example of icon use is for compliance to certain Digital Advertising Alliance (DAA) self-regulatory principles for Online Behavioral Advertising (OBA). If you would like more information about the OBA Self Regulation program, please visit http://www.aboutads.info.

The video player can use the `adIcons` property to avoid displaying duplicate icons over any icons that might be provided in the ad unit. Until the industry provides more guidance on how to pass metadata using common ad-serving protocols, this property is limited to a Boolean response. The default value is `False`. If one or more ad icons are present within the ad, the value returned is `True`. When set to `True`, the video player should not display any ad icons of its own.

3.3 Dispatched Events

All events are dispatched from the ad unit to the video player, usually in response to method calls made by the video player. Event handlers in the video player may in turn call ad methods. Video players must be written in a way to prevent a stack overflow caused by calling back into methods as a result of an event.

The ad unit should avoid dispatching any events before `AdLoaded` is sent or after `AdStopped` or `AdSkipped` is sent.

3.3.1 AdLoaded

When the video player calls the `initAd()` method, the ad unit can begin loading assets. Once loaded and ready for display, the ad dispatches the `AdLoaded` event. No UI elements should be visible before `AdLoaded` is sent, but sending `AdLoaded` indicates that the ad unit has verified that all files are ready to execute. Also, if `initAd()` was called, and the ad unit is unable to display and/or send `AdLoaded`, then `AdError` should be dispatched.

3.3.2 AdStarted

The `AdStarted` event is sent by the ad unit to notify the video player that the ad is displaying and is a response to the `startAd()` method.

3.3.3 AdStopped

The `AdStopped` event is sent by the ad unit to notify the video player that the ad has stopped displaying and all ad resources have been cleaned up. This event is only for responding the `stopAd()` method call made by the video player. It should never be used to initiate the ad unit's end or used to inform the video player that it can now call `stopAd()`.

3.3.4 AdSkipped

The `AdSkipped` event is sent by the ad unit to notify the video player that the ad has been skipped, stopped displaying and all ad resources have been cleaned up.

The `AdSkipped` event can be sent in response to the `skipAd()` method call or as a result of a skip control activated within the ad unit (rather than in the video player).

In response to a `skipAd()` method call, the ad unit must stop ad play, clean up all resources, and send the `AdSkipped` event. If a skip control is activated within the ad unit, the ad unit must stop ad play, clean up all resource, and send the `AdSkipped` event followed by the `AdStopped` event. Sending the `AdStopped` event for skip controls activated in the ad unit ensures that video players using earlier versions of VPAID receive notice that the ad has stopped playing.

3.3.5 AdSkippableStateChange

`AdSkippableStateChange` is new to VPAID 2.0.

When an ad unit only allows its creative to be skipped within a specific time frame, it can use the `AdSkippableStateChange` event to prompt the video player to check the value of the `adSkippableState` property, which keeps the video player updated on when the ad can be skipped and when it cannot be skipped.

3.3.6 AdSizeChange

`AdSizeChange` is new to VPAID 2.0

The `AdSizeChange` event is sent in response to the `resizeAd()` method call. When the video player resizes, it notifies the ad unit so that the ad unit can also scale to maintain the same ad space ratio that it had relevant to the previous video player size.

When the video player calls `resizeAd()`, the ad unit must scale its width and height value to equal or less than the width and height value supplied in the video player call. If the video player doesn't provide width and height values (as in fullscreen mode), then the ad unit can resize to any dimension.

Once the ad unit has resized itself, it writes width and height values to the `adWidth` and `adHeight` properties, respectively. The `AdSizeChange` event is then sent to confirm that the ad unit has resized itself.

See resizeAd(), adWidth and adHeight for more information.

3.3.7 AdLinearChange

The `AdLinearChange` event is sent by the ad unit to notify the video player that the ad unit has changed playback mode. To find out the current state of the ad unit's linearity, the video player must use the get `adLinear` property and update its UI accordingly. See the adLinear property for more information.

3.3.8 AdDurationChange

`AdDurationChange` is new to VPAID 2.0.

The duration for some video ads can change in response to user interaction or other factors. When the ad duration changes, the ad unit updates the values of the `adDuration` and `adRemainingTime` properties and dispatches the `AdDurationChange` event, notifying the video player that duration has changed. The video player can then get `adDuration` and `adRemainingTime` to update its UI, such as the duration indicator, if applicable.

During normal playback, `adDurationChange` should not be dispatched unless the total duration of the ad changes.

3.3.9 AdExpandedChange

When the expanded state of the ad changes, the ad unit must update the `adExpanded` property and dispatch the `AdExpandedChange` event to notify the video player of the change. The video player responds by using the get `adExpanded` property to update its UI accordingly. An `AdExpandedChange` event may be triggered by the `expandAd()` method.

The AdExpandedChange event is only for notifying the player of a change in ad unit expansion, such as the expand or collapse of an interactive panel. To dispatch a change in standard display size, please use AdSizeChange.

3.3.10 AdRemainingTimeChange (Deprecated in 2.0)

The `AdRemainingTimeChange` event is still supported in order to accommodate ads and video players using VPAID 1.0; however, in 2.0 versions, please use `AdDurationChange`.

The `AdRemainingTimeChange` event is sent by the ad unit to notify the video player that the ad's remaining playback time has changed. The video player may get the `adRemainingTime` property and update its UI accordingly.

Upon initial duration change, the ad unit should update the `adRemainingTime` property and send the `AdRemainingTimeChange` event at least once per second but no more than four times per second (to maintain optimum performance) so that the video player can keep its UI in synch with actual time remaining.

3.3.11 AdVolumeChange

If the ad unit supports volume, any volume changes are updated in the `adVolume` property and the `AdVolumeChange` event is dispatched to notify the video player of the change. The video player may then use the get `adVolume` property and update its UI accordingly.

3.3.12 AdImpression

The `AdImpression` event is used to notify the video player that the user-visible phase of the ad has begun. The `AdImpression` event may be sent using different criteria depending on the type of ad format the ad unit is implementing.

For a linear mid-roll ad, the impression should coincide with the AdStart event. However, for a non-linear overlay ad, the impression will occur when the invitation banner is displayed, which is normally before the ad video is shown. This event matches that of the same name in *Digital Video In-Stream Ad Metrics Definitions*, and must be implemented to be IAB compliant.

3.3.13 AdVideoStart, AdVideoFirstQuartile, AdVideoMidpoint, AdVideoThirdQuartile, AdVideoComplete

These five events are sent by the ad unit to notify the video player of the ad unit's video progress and are used in VAST under the same event names. Definitions can be found under "Percent complete" events in *Digital Video In-Stream Ad Metrics Definitions.* These events must be implemented for ads to be IAB compliant, but only apply to the video portion of the ad experience, if any.

3.3.14 AdClickThru

The `AdClickThru` event is sent by the ad unit when a clickthrough occurs. Three parameters can be included to give the video player the option for handling the event.

Three parameters are available for the event:

- **String url:** enables the ad unit to specify the clickthrough url
- **String Id:** used for tracking purposes
- **Boolean playerHandles:** indicates whether the video player or the ad unit handles the event. Set to true, the video player opens the new browser window to the URL provided. Set to false, the ad unit handles the event.

The `AdClickThru` event is included under the same name in *Digital Video In-Stream Ad Metrics Definitions* and must be implemented to be IAB compliant.

3.3.15 AdInteraction

`AdInteraction` is new in VPAID 2.0.

This event was introduced to capture all user interactions under one metric aside from any clicks that result in redirecting the user to specified site. `AdInteraction` events might include hover-overs, clicks that don't result in a `ClickThru`, click-and-drag interactions, and the events described in section 3.3.16. While `AdInteraction` does not replace any other metrics, it can be used in addition to other metrics. Keep in mind that recording both an `AdUserMinimize` and an `AdInteraction` for the same event is just one event with two names. Other custom interactions, such as "Dealer Locator" for example don't exist in any VPAID events, so it could be recorded under the `AdInteraction` event.

The `AdInteraction` event is sent by the ad unit to indicate any interaction with the ad EXCEPT for ad clickthroughs. An ad clickthrough is indicated using the `AdClickThru` event described in section 3.3.14.

One parameter is available for the event:

- **String Id:** used for tracking purposes

3.3.16 AdUserAcceptInvitation, AdUserMinimize, AdUserClose

The `AdUserAcceptInvitation`, `AdUserMinimize` and `AdUserClose` events are sent by the ad unit when they meet requirements of the same names as set in *Digital Video In-Stream Ad Metrics Definitions*. Each of these events indicates user-initiated action that the ad unit dispatches to the video player. The video player may choose to report these events externally, but takes no other action.

3.3.17 AdPaused, AdPlaying

The `AdPaused` and `AdPlaying` events are sent in response to the `pauseAd()` and `resumeAd()` method calls, respectively, to confirm that the ad has either paused or is playing. Sending `AdPaused` indicates that the ad has stopped all audio and any animation in progress. Other settings, such as adjusting the ad's visibility or removing ad elements from the UI, may be implemented until `resumeAd()` is called. Sending `AdPlaying` indicates that the ad unit has resumed playback from the point at which it was paused. See

pauseAd() and resumeAd() method descriptions for more detail.

3.3.18 AdLog

The `AdLog` event is optional and can be used to relay debugging information.

One parameter is available for this event:

- **String Id:** used for tracking purposes

3.3.19 AdError

The `AdError` event is sent when the ad unit has experienced a fatal error. Before the ad unit sends `AdError` it must clean up all resources and cancel any pending ad playback. The video player must remove any ad UI, and recover to its regular content playback state. The **String message** parameter can be used to provide more specific information to the video player.

3.4 Error Handling and Timeouts

Any fatal errors detected by the ad unit initiates an `AdError` event that cancels ad playback and returns the video player to content playback state. Errors should only be sent through the `AdError` event and *no uncaught exceptions should ever be thrown.*

The video player should also implement timeout mechanisms anywhere an ad response is expected and not received.

Recovery actions are provided for each of the timeout situations below:

- **Delay of request of ad file to ad file successfully loaded:** cancel ad load and move on to the next ad or return to regular content playback.
- **Calling** `initAd()` **and not receiving** `AdLoaded` **event:** remove ad from UI and delete all ad resources. Move on to next ad or return to regular content playback.
- **Calling** `startAd()` **and not receiving** `AdStarted` **event:** call `stopAd()`. If no response is received, remove the ad from the UI and delete all resources. Move on to the next ad or return to regular content playback.
- **Receiving** `AdLinearChange` **event with** `adLinear` **set to true but never receiving** `AdLinearChange` **event with** `adLinear` **set to false (excluding ad pause/resume time):** call `stopAd()`. If no response is received, remove the ad from the UI and delete all resources. Move on to the next ad or return to regular content playback.
- **Calling** `stopAd()` **and not receiving** `AdStopped` **event:** remove the ad from the UI and delete all resources. Move on to the next ad or return to regular content playback.

The amount of time to allot for a timeout is up to the publisher and should be discussed with partners and vendors, but allowing more than a few seconds before moving on may cause the publisher to miss opportunities for executing alternative ads within the allotted ad space.

4 Security

VPAID was designed to allow for unidirectional scripting, where possible. All VPAID methods and properties are on the VPAID member of the ad object, allowing scripting from the video player to the ad unit. However, the ad unit can only send notification of an event to the video player. For client platforms such as Flash, this allows unidirectional scripting from video player to ad unit but prevents the ad unit from scripting into the video player or using a JavaScript bridge to attack the web page.

In some cases a publisher may allow scripting to the video player using ExternalInterface (allowScriptAccess="always"). Allowing script access may be used to support verification services because it allows for transparency around the context in which the ad is being served. However, allowing scripting from the ad to the video player can cause discrepancies in publisher data and poses a security risk. Publishers and advertisers should discuss script access as part of campaign setup.

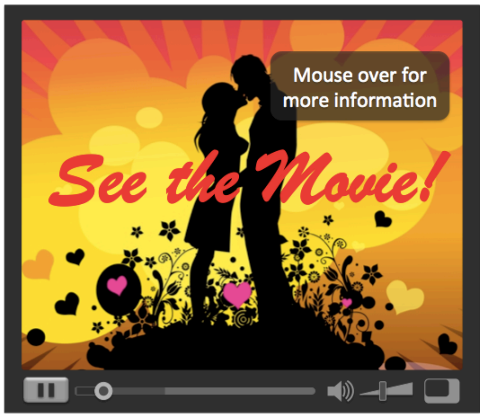
For each implementation technology represented in sections 6-8, a security subsection provides details necessary for secure implementation.

5 Example Ads and the VPAID Process

To show how VPAID works in different video ad types, we've represented the VPAID process for four common ad types:

- Clickable Pre-Roll
- Companion Banner (achieved without VPAID)
- Overlay Banner
- Overlay Banner with Click-to-Linear Video Ad

5.1 Clickable Pre-Roll



Clickable pre-roll ads are used to enable an interactive overlay for a video ad served in advance of video content. In a clickable pre-roll ad, the viewer can interact with the overlay image or text and ultimately be redirected to the advertiser's specified webpage.

VPAID Process for a Clickable Pre-Roll Ad:

1. Video player calls handshakeVersion("2.0").
    a. Ad unit returns "2.0."
2. Video player calls initAd(500, 400, "normal", 500).
    a. Ad unit loads and returns `AdLoaded`.
3. Video player calls `startAd()`.
    a. Ad unit starts ad display and returns `AdStarted`.
4. Video player gets `adLinear`.
    a. Video player sees that the value is `true`.
    b. Video player delays loading/displaying content video.
5. Video player gets `adDuration`, sets `adVolume`, manages playback with calls to `pauseAd()`, `resumeAd()`, `resizeAd()`, etc. *
    a. Ad unit responds appropriately to each call.
    b. When ad play is complete, ad unit sends `AdStopped`.
6. Video player buffers and plays content video.

* Without VPAID, the interactivity offered in step 5 is only possible with a custom solution. The `adDuration` property in VPAID supports variable duration resulting from interaction. The `resizeAd()` method ensures predictable ad size scaling. And the set `adVolume` property enables control over ad volume.

5.2 Companion Banner

The companion ad format is used with linear ads to provide an interactive ad outside the video player (shown below the video player in the example to the right).

The companion banner information can be provided either in the initial VAST response, or by using the VPAID `adCompanions()` property. See section 3.2.9 for more information.

See the Movie!
See the Movie!

5.3 Overlay Banner

An overlay banner is an ad that displays over content video during video playback. The video player may control the designated ad display area dimensions, position and display timing.

See the Movie!
Feel the Warmth!

VPAID Process for Overlay Banners:

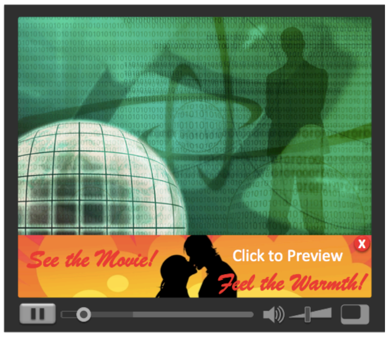
1. Video player calls handshakeVersion("2.0").
   a. Ad unit returns "2.0."
2. Video player calls initAd(500, 400, "normal", 500).
   a. Ad unit loads and returns `AdLoaded`.
3. Video player calls `startAd()`.
   a. Ad unit starts ad display and returns `AdStarted`.
4. Video player gets `adLinear`.
   a. Video player sees that the value is `false`.
   b. Video player begins loading/displaying content video.
   c. Video player may begin playing content video when ready.
5. Video player calls `stopAd()` when ad display time has elapsed.
   a. Ad unit stops ad play, cleans up resources and responds with `AdStopped`.
6. Content video continues to play.

Without VPAID, the overlay banner can be executed using a non-linear ad with defined duration in which the video player defines ad position. However, interaction is limited unless

VPAID or some custom solution for the ad unit and video player is implemented. The example below shows how VPAID enables interactivity for the overlay banner.

5.4 Overlay Banner with Click-to-Linear Video Ad

VPAID enables a rich, interactive experience in video overlay banners. As in the standard overlay example on the previous page, the ad displays over the content video while the content video is playing. However, VPAID enables the video player to stop its video content playback when a viewer clicks the ad so that the ad unit can initiate a linear video ad or expand for a broader engagement opportunity. When the linear ad or expanded portion has completed, the content video resumes playback.



VPAID Process for a Click-to-Linear Video Ad:

1. Video player calls handshakeVersion("2.0").
    a. Ad unit returns "2.0."
2. Video player calls initAd(500, 400, "normal", 500).
    a. Ad unit loads and returns `AdLoaded`.
3. Video player calls `startAd()`.
    a. Ad unit starts ad display and returns `AdStarted`.
4. Video player gets `adLinear`.
    a. Video player sees that the value is `false`.
    b. Video player delays loading content video.
5. If viewer clicks to expand the ad:
    a. Ad unit sets `adLinear = true` and sends `AdLinearChange`.
    b. Ad may set `adExpanded = true` and send `AdExpandedChange`
    c. Ad sends `AdInteraction`.
    d. Video player pauses content video.
    e. The video ad plays and completes.
    f. Ad unit sets adExpanded = false and sends AdExpandedChange.
    g. Ad unit sets `adLinear = false` and sends `AdLinearChange`.
    h. Video player continues content video playback
6. Video player calls `stopAd()` when ad display time has elapsed
    a. Ad unit sends `AdStopped`.
    b. Video player continues content video playback.

Without VPAID (or a custom solution), this interaction is not possible. Other interactions, such expanding the ad upon click and variable duration dependent on interaction, are also enabled with VPAID. Creative video ad development is encouraged with VPAID as long as guidelines are respected along with other applicable digital video guidelines listed on page 7.

6 ActionScript 3 Implementation

This section provides information for how to implement VPAID using ActionScript 3. Also included are details on using custom events and addressing security concerns.

6.1 API Specifics

The loaded ad .swf file should be loaded into its own `ApplicationDomain` and therefore be treated as a *data type. Both the ad unit and the video player need their own VPAID interface (IVPAID) definition since they both run in separate `ApplicationDomains`. The interface cannot be shared.

The video player accesses VPAID by calling `getVPAID` on the ad object. For safety when calling VPAID, the video player may wrap the returned *object with a class that implements VPAID explicitly, as shown in the following example. The VPAID class must extend `EventDispatcher`, and the video player should call `addEventHandler` on the VPAID object using events with String names for the events listed in section 3.3.

To provide frame rate information (`Stage.frameRate`), use the `environmentVars` paramenter in the `initAd()` method.

For more information on passing custom event information, please Custom Events in section 6.2 following the sample code below.

Sample code for implementing VPAID in ActionScript 3:

```
package
{
 public interface IVPAID
 {
  // Properties
  function get adLinear() : Boolean;
  function get adWidth() : Number
  function get adHeight() : Number
  function get adExpanded() : Boolean;
  function get adSkippableState() : Boolean;
  function get adRemainingTime() : Number;
  function get adDuration() : Number;
  function get adVolume() : Number;
  function get adCompanions() : String;
  function get adIcons() : Boolean;
  // Methods
  function handshakeVersion(playerVPAIDVersion : String) : String;
  function initAd(width : Number, height : Number, viewMode : String, desiredBitrate : Number, creativeData :
String="", environmentVars : String="") : void;
  function resizeAd(width : Number, height : Number, viewMode : String) : void;
```

```
    function startAd() : void;
    function stopAd() : void;
    function pauseAd() : void;
    function resumeAd() : void;
    function expandAd() : void;
    function collapseAd() : void;
    function skipAd() : void;
  }
}

package
{
  // Player wrapper for untyped loaded swf
  public class VPAIDWrapper extends EventDispatcher implements IVPAID
  {
    private var _ad:*;

    public function VPAIDWrapper(ad:*) {
        if(ad.hasOwnProperty('getVPAID'))
            _ad = ad.getVPAID();
        else
      _ad = ad;
    }

    // Properties
    public function get adLinear():Boolean {
      return _ad.adLinear;
    }
    public function get adWidth():Number {
      return _ad.adWidth;
    }
    public function get adHeight():Number {
      return _ad.adHeight;
    }
    public function get adExpanded():Boolean {
      return _ad.adExpanded;
    }
    public function get adSkippableState():Boolean {
      return _ad.adSkippableState;
    }
    public function get adRemainingTime():Number {
      return _ad.adRemainingTime;
    }
    public function get adDuration():Number{
      return _ad.adDuration;
    }
    public function get adVolume():Number {
      return _ad.adVolume;
    }
    public function set adVolume(value:Number):void {
      _ad.adVolume = value;
    }
    public function get adCompanions():String{
      return _ad.adCompanions;
    }

    // Methods
    public function handshakeVersion(playerVPAIDVersion : String):String {
      return _ad.handshakeVersion(playerVPAIDVersion);
    }
    public function initAd(width:Number, height:Number, viewMode:String, desiredBitrate:Number,
creativeData:String="", environmentVars : String=""):void {
      _ad.initAd(width, height, viewMode, desiredBitrate, creativeData, environmentVars);
    }
    public function resizeAd(width:Number, height:Number, viewMode:String):void {
      _ad.resizeAd(width, height, viewMode);
    }
    public function startAd():void {
      _ad.startAd();
    }
    public function stopAd():void {
      _ad.stopAd();
    }
    public function pauseAd():void {
      _ad.pauseAd();
    }
    public function resumeAd():void {
      _ad.resumeAd();
    }
    public function expandAd():void {
      _ad.expandAd();
    }
    public function collapseAd():void {
      _ad.collapseAd();
    }
    public function skipAd():void {
      _ad.collapseAd();
    }

    // EventDispatcher overrides
    override public function addEventListener(type:String, listener:Function, useCapture:Boolean=false,
priority:int=0, useWeakReference:Boolean=false):void {
      _ad.addEventListener(type, listener, useCapture, priority, useWeakReference);
    }
    override public function removeEventListener(type:String, listener:Function,
useCapture:Boolean=false):void {
      _ad.removeEventListener(type, listener, useCapture);
    }
    override public function dispatchEvent(event:Event):Boolean {
      return _ad.dispatchEvent(event);
    }
    override public function hasEventListener(type:String):Boolean {
      return _ad.hasEventListener(type);
    }
    override public function willTrigger(type:String):Boolean {
      return _ad.willTrigger(type);
    }
  }
```

}

6.2 Custom events

As with the VPAID interface itself, event class definitions are not shared between the video player and the ad unit. The following class definition can be defined in both the video player and the ad unit, but since they are in separate ApplicationDomains it cannot be shared.

The ad unit must create a flash.events.Event derived class that implements a data property getter as shown in the example below.

Two considerations should be noted when implementing VPAID:

- When defining the VPAIDEvent object, the "bubbles" attribute should be set to false (as in the public function VPAIDEvent() near the bottom of the example provided).
- When listening for a VPAIDEvent, the listener can optionally call "stopPropagation".

```
package
{
 import flash.events.Event;

 public class VPAIDEvent extends Event
 {
  public static const AdLoaded : String = "AdLoaded";
  public static const AdStarted : String = "AdStarted";
  public static const AdStopped : String = "AdStopped";
  public static const AdSkipped : String = "AdSkipped";
  public static const AdLinearChange : String = "AdLinearChange";
  public static const AdSizeChange : String = "AdSizeChange";
  public static const AdExpandedChange : String = "AdExpandedChange";
  public static const AdSkippableStateChange : String = "AdSkippableStateChange";
  public static const AdRemainingTimeChange : String = "AdRemainingTimeChange";
  public static const AdDurationChange : String = "AdDurationChange";
  public static const AdVolumeChange : String = "AdVolumeChange";
  public static const AdImpression : String = "AdImpression";
  public static const AdVideoStart : String = "AdVideoStart";
  public static const AdVideoFirstQuartile : String = "AdVideoFirstQuartile";
  public static const AdVideoMidpoint : String = "AdVideoMidpoint";
  public static const AdVideoThirdQuartile : String = "AdVideoThirdQuartile";
  public static const AdVideoComplete : String = "AdVideoComplete";
  public static const AdClickThru : String = "AdClickThru";
  public static const AdInteraction : String = "AdInteraction";
  public static const AdUserAcceptInvitation : String = "AdUserAcceptInvitation";
  public static const AdUserMinimize : String = "AdUserMinimize";
  public static const AdUserClose : String = "AdUserClose";
  public static const AdPaused : String = "AdPaused";
  public static const AdPlaying : String = "AdPlaying";
  public static const AdLog : String = "AdLog";
  public static const AdError : String = "AdError";

  private var _data:Object;
```

```
    public function VPAIDEvent(type:String, data:Object=null, bubbles:Boolean=false,
cancelable:Boolean=false) {
      super(type, bubbles, cancelable);
      _data = data;
    }
    public function get data():Object {
      return _data;
    }
    public function clone():Object {
      return new VpaidEvent(type, data, bubbles, cancelable);
    }
}
}
// sample ad dispatch call from a function within ad's VPAID class
dispatchEvent(new VPAIDEvent(VPAIDEvent.AdStarted));
dispatchEvent(new VPAIDEvent(VPAIDEvent.AdClickThru,
  {url:myurl,id:myid,playerHandles:true}));
```

The video player uses addEventListener with a handler function that that receives an Object as a parameter. To continue the above example:

```
public function onAdClickThru(event:Object) : void
{
  trace("Ad url is: " + event.data.url);
}
_VPAID.addEventListener(VPAIDEvent.AdClickThru, onAdClickThru);
```

6.3 Security

To implement unidirectional scripting in ActionScript 3, use Security.allowDomain("<playerdomain or *>"). The ad swf must also be served from a domain where /crossdomain.xml allows the ad swf to be loaded by the video player domain or *. The video player should load the ad swf into a separate security and application domain. The ad unit SWF should not access the Stage. Special attention should be paid in the ad unit SWF for handling ads that require JavaScript access via ExternalInterface: an `AdError` should be thrown if this access is not allowed.

7 Silverlight Implementation

7.1 API Specifics

The Silverlight ad unit will expose the above APIs to support the communication from the video player. As it's not very easy to access getVPAID method on an ad unit, the ad unit will implement "IVPAID" interface. The video player can determine if the ad unit implements "IVPAID" interface and if it does, will understand that the ad unit is a VPAID compliant ad. A video player/ad unit combination may decide that using VPAID is unsafe, but the recommended way would be to use a type safe interface, which defines the above APIs. This type safety can be achieved by using the binary IVPAID interface that the ad unit can implement. The video player will also use this binary interface to check if the ad unit implements the interface and thus can be sure that the ad unit is a VPAID compliant ad. More information about this interface can be found athttp://www.iab.net/iab_products_and_industry_services/508676/508950/vpaid).

Following is the interface which VPAID-compliant Silverlight ads will implement (based on above explanation of VPAID.

```
public interface IVPAID {

        #region VPAID Methods

        string handshakeVersion(string version);
        void initAd(uint width, uint height, string viewMode, int desiredBitrate, string creativeData="",
string environmentVars="");
        void startAd();
        void stopAd();
        void resizeAd(uint width, uint height, string viewMode);
        void pauseAd();
        void resumeAd();
        void expandAd();
        void collapseAd();
        void skipAd();

        #endregion

        #region VPAID Properties

        bool adLinear { get; }
        unit adWidth { get; }
        unit adHeight { get; }
        bool adExpanded { get; }
        bool adSkippableState { get; }
        TimeSpan adRemainingTime { get; }
        TimeSpan adDuration { get; }
        string adCompanions { get; }
        float adVolume { get; set; }
        bool adIcons { get; }
        #endregion

        #region VPAID Events

        event EventHandler AdLoaded;
        event EventHandler AdStarted;
        event EventHandler AdStopped;
        event EventHandler AdSkipped;
        event EventHandler AdPaused;
        event EventHandler AdSizeChange;
        event EventHandler AdPlaying;
        event EventHandler AdExpandedChange;
        event EventHandler AdSkippableStateChange;
        event EventHandler AdLinearChange;
        event EventHandler AdVolumeChange;
        event EventHandler AdVideoStart;
        event EventHandler AdVideoFirstQuartile;
        event EventHandler AdVideoMidPoint;
        event EventHandler AdVideoThirdQuartile;
        event EventHandler AdVideoComplete;
        event EventHandler AdUserAcceptInvitation;
        event EventHandler AdUserClose;
        event EventHandler AdUserMinimize;
        event EventHandler<ClickThroughArgs> AdClickThru;
        event EventHandler<AdInteraction Args> AdInteraction;
        event EventHandler<StringEventArgs> AdError;
        event EventHandler<StringEventArgs> AdLog;
        event EventHandler AdDurationChange;
        event EventHandler AdRemainingTimeChange;
        event EventHandler AdImpression;

        #endregion
}
```

7.2 Custom Events

```
#region Event Argument classes

public class StringEventArgs : EventArgs {
        string Message { get; set; }
}

public class ClickThroughArgs : EventArgs {
        public string Url { get; set; }
        public string Id { get; set; }
        public bool PlayerHandles { get; set; }
}

#endregion
```

7.3 Security

The Silverlight ad XAP must be served from a domain where policy file allows the ad XAP to be loaded by the video player domain or *.

8 JavaScript Implementation

This section outlines JavaScript-based VPAID APIs that can be used by video player and in-stream video ads to interact in a standard way, enabling interoperability between various video player and ad unit implementations for HTML5 capable devices. The Javascript APIs follow the same basic video player and ad unit interaction approach as of Flash™ and Silverlight™ APIs. For example:

1. The video player is responsible for making the ad call and parsing of the ad xml response from the ad server
2. The video player invokes a set of functions provided by the ad VPAID object
3. The ad unit fires events/callbacks for the major events in the ad unit and the video player acts on them
4. The video player provides the ad unit with the location where it is supposed to render itself in

All effort is made to keep the APIs, property, and event definitions consistent with the flash and Silverlight specification, but there are few differences as mandated by the JavaScript language and environment. These differences are described in the following section.

8.1 API Specifics

8.1.1 Methods

initAd()

initAd(width : Number, height : Number, viewMode : String, desiredBitrate : Number, creativeData : Object, environmentVars : Object) : void

- The basic use of `initAd()` and parameter is same as the AS3 and silverlight implementation, but there are few differences:
  - The 'creativeData' and 'environmentVars' arguments are Objects
  - creativeData parameter is used for passing in additional ad initialization data, specifically adParameters node of a *VAST* response.
  - The 'environmentVars' object contains a reference, 'slot', to the HTML element on the page in which the ad is to be rendered. The ad unit essentially gets control of that element. See the following code example on how to set this value.
  - The 'environmentVars' object would contain a reference, 'videoSlot', to the video element on the page where the ad video is to be rendered and a boolean flag, 'videoSlotCanAutoPlay' indicating whether the 'videoSlot' is capable of autoplaying. It is upto the player implementation to decide what video element to pass to the ad. Two choices with associated tradeoffs are:
    1. The player can pass the video element used for playing the content video.

   - This allows the ad unit and content videos to share the same video element, allowing a consistent user experience between content and ad playback including sharing of the control bar.
   - This approach is similar to flash implementation.
   - This also allows the ad implementer to get around the autoplay limitation in iOS and Chrome mobile where video elements can't start without user interaction
   - In this case, boolean flag , 'videoSlotCanAutoPlay', should be set to true.
2. The player can pass a video element different than what it used to play the content video.
   - The advantage of this choice is that player can pre-buffer the video content to be played after the ad, potentially providing a better content play experience.
   - In this case, on iOS and Chrome mobile, the ad video will not autoplay. For these devices, and any future devices with similar restriction, boolean flag , 'videoSlotCanAutoPlay', should be set to false.
   - For the cases where ad video cannot autoplay, it is recommended that the ad video be rendered with "poster" attribute and/or the ad video is built such that the first frame is meaningful to the user.

Example code for populating 'CreativeData' and 'EnvironmentVars':

```
// Populating CreativeData - The AdParameters string should be extracted by parsing the VAST XML
CreativeData.AdParameters = "Linear/NonLinear AdParameter node from VAST without the
CDATA wrapper";

  // Example assumes that ad is to be rendered within a div with id 'videoAdLayer'
  // Example assumes that the content video plays within video element with id 'videoElement'
  var slot = document.getElementById('videoAdLayer');
  environmentVars.slot = slot;
  // Share the content video slot with the ad   environmentVars.videoSlot =
document.getElementById('videoElement');
  environmentVars.videoSlotCanAutoPlay = true;
  ad.initAd(slot.style.width, slot.style.height, viewMode, desiredBitrate, creativeData,
environmentVars)
```

// The following method call is specific to JavaScript

subscribe

subscribe (fn : Reference, event : String, [listenerScope : Reference]) :void

- The video player calls this method to register a listener to a particular event
  - fn is a reference to the function that needs to be called when the specified event occurs
  - event is the name of the event that the video player is subscribing to
  - [optional] listenerScope is a reference to the object in which the function is defined

unsubscribe

unsubscribe (fn: Reference, event : String) :void

- The video player calls this method to remove a listener for a particular event
    - event is the name of the event that the video player is unsubscribing from
    - fn is the event listener that is being removed

8.1.2 Properties

The ad maintains properties described as part of the section 3.2. For the JavaScript implementation, the ad provides a getter, and if the property is settable, a setter function for each of the proprties. Player uses property specific getter and setter functions to access the ad properties. For example, player gets/sets adVolume using the following functions implemented by the ad:

- getAdVolume() : Number
- setAdVolume(val : Number) : void

If the property can only be read by the player but cannot be set by it, ad only provides a getter function. For example for adExpanded property, ad only provides following getter function:

- getAdExpanded() : Number

Same scheme is used for all of the other properties described in section 3.2.

8.1.3 Security

VPAID uses the ad rendering approach as described in the "IAB Best Practice for Rich Media Ads in Asynchronous Ad Documents" found at the following link for further review:

http://www.iab.net/media/file/rich_media_ajax_best_practices.pdf

The VPAID ad unit is rendered within a friendly IFRAME (FIF) and can access the DOM of the page in which the player is rendered.

As modern technology further enables isolation between the display ad and the publisher page content, IAB will embark on developing specifications for enhanced security and VPAID will be updated to incorporate applicable security recommendations.

8.1.4 Ad Rendering and Namespace Management

To provide interoperability between the video player and ad unit implementations, the video player and the ad unit must follow, beyond the API specification, guidelines related to ad loading and rendering. The ad loading and rendering guidelines are described below and include example code.

- In order to avoid namespace collisions across multiple ads on the same page, all ad javascript should be loaded in a friendly iframe, effectively making the ad javascript the only element in the new DOM
- A standardized global factory function named 'getVPAIDAd()' will be called in the iframe to get an object of the ad and returned to the video player on the content page.

- Any supporting javascript required by the ad unit would need to be loaded by the ad in the friendly iframe.
- This is the same approach as recommended by the IAB Best Practice for Rich Media Ads in Asynchronous Ad Documents.

8.1.5 Examples

Example of AdLoading

```
iframe = document.createElement('iframe');
iframe.id = "adloaderframe";
document.body.appendChild(iframe);
// 'url' points to the ad js file
iframe.contentWindow.document.write('<script src="' + url + '"></scr' + 'ipt>');
var fn = iframe.contentWindow['getVPAIDAd'];
if (fn && typeof fn == 'function') {
   VPAIDCreative = fn();
}
```

In the example the return value of the function is an instance of the ad class

Example code to check if the VPAIDCreative implements all of the functions required by the VPAID spec

```
          this.checkVPAIDInterface = function(VPAIDCreative) {
            if(
                VPAIDCreative.handshakeVersion && typeof
VPAIDCreative.handshakeVersion == "function" && VPAIDCreative.initAd && typeof
VPAIDCreative.initAd == "function" &&
                VPAIDCreative.startAd && typeof VPAIDCreative.startAd == "function" &&
                VPAIDCreative.stopAd && typeof VPAIDCreative.stopAd == "function" &&
                VPAIDCreative.skipAd && typeof VPAIDCreative.skipAd == "function" &&
                VPAIDCreative.resizeAd && typeof VPAIDCreative.resizeAd == "function" &&
                VPAIDCreative.pauseAd && typeof VPAIDCreative.pauseAd == "function" &&
                VPAIDCreative.resumeAd && typeof VPAIDCreative.resumeAd == "function"
&&
                VPAIDCreative.expandAd && typeof VPAIDCreative.expandAd == "function"
&&
                VPAIDCreative.collapseAd && typeof VPAIDCreative.collapseAd == "function"
&&
                VPAIDCreative.subscribe && typeof VPAIDCreative.subscribe == "function" &&
                VPAIDCreative.unsubscribe && typeof VPAIDCreative.unsubscribe ==
"function" ){
              return true;
            }
            return false;
          };
```

Example Ad Implementation

This is an example of basic implementation of a Linear VPAID ad.

```
LinearAd = function() {
  // The slot is the div element on the main page that the ad is supposed to occupy
  this._slot = null;
  // The video slot is the video object that the creative can use to render and video element it
might have.
  this._videoSlot = null;
  };   LinearAd.prototype.initAd = function(width, height, viewMode, desiredBitrate,
    creativeData, environmentVars) {
    // slot and videoSlot are passed as part of the environmentVars
    this._slot = environmentVars.slot;
    this._videoSlot = environmentVars.videoSlot;

    console.log("initAd");
  };

  LinearAd.prototype.startAd = function() {
    console.log("Starting ad");
    .
    .
    .
  };
  LinearAd.prototype.stopAd = function(e, p) {
    console.log("Stopping ad");
    .
    .
    .
  };
  LinearAd.prototype.setAdVolume = function(val) {
    console.log("setAdVolume");
    .
    .
    .
  };

  LinearAd.prototype.getAdVolume = function() {
    console.log("getAdVolume");
    .
    .
    .
  };
  LinearAd.prototype.resizeAd = function(width, height, viewMode) {
    console.log("resizeAd");
    .
    .
    .
  };
  LinearAd.prototype.pauseAd = function() {
    console.log("pauseAd");
    .
    .
    .
  };
  LinearAd.prototype.resumeAd = function() {
    console.log("resumeAd");
    .
    .
    .
  };
  LinearAd.prototype.expandAd = function() {
    console.log("expandAd");
    .
    .
    .
  };
  LinearAd.prototype.getAdExpanded = function(val) {
    console.log("getAdExpanded");
    .
    .
    .
  };

  LinearAd.prototype.getAdSkippableState = function(val) {
    console.log("getAdSkippableState");
    .
    .
    .
  };

  LinearAd.prototype.collapseAd = function() {
    console.log("collapseAd");
    .
    .
    .
  };

  LinearAd.prototype.skipAd = function() {
    console.log("skipAd");
```

```
    .
    .
    .
  };

      // Callbacks for events are registered here
  LinearAd.prototype.subscribe = function(aCallback, eventName, aContext) {
    console.log("Subscribe");
    .
    .
    .
  };
      // Callbacks are removed based on the eventName
  LinearAd.prototype.unsubscribe = function(eventName) {
    console.log("unsubscribe");
    .
    .
    .
  }
getVPAIDAd = function() {
  return new LinearAd();
};
```

Sample Video player Interface Code

- This code is meant to be part of the video player that interacts with the ad.
- It effectively wraps the VPAID creative into an enveloping object that the video player can interact with
  - The example illustrates how callback can be registered with the creative.

```
// This class is meant to be part of the video player that interacts with the Ad.
// It takes the VPAID creative as a parameter in its contructor.
VPAIDWrapper = function(VPAIDCreative) {
            this._creative = VPAIDCreative;
            if(!this.checkVPAIDInterface(VPAIDCreative))
            {
              //The VPAIDCreative doesn't conform to the VPAID spec
              return;
            }
            this.setCallbacksForCreative();
          // This function registers the callbacks of each of the events

          VPAIDWrapper.prototype.setCallbacksForCreative = function() {

            //The key of the object is the event name and the value is a reference to the
callback function that is registered with the creative
```

```
        var callbacks = {
          AdStarted : this.onStartAd,
          AdStopped : this.onStopAd,
          AdSkipped : this.onSkipAd,
          AdLoaded : this.onAdLoaded,
          AdLinearChange : this.onAdLinearChange,
          AdSizeChange : this.onAdSizeChange,
          AdExpandedChange : this.onAdExpandedChange,
          AdSkippableStateChange : this.onAdSkippableStateChange,
          AdDurationChange : this.onAdDurationChange,
          AdRemainingTimeChange : this.onAdRemainingTimeChange,
          AdVolumeChange : this.onAdVolumeChange,
          AdImpression : this.onAdImpression,
          AdClickThru : this.onAdClickThru,
          AdInteraction : this.onAdInteraction,
          AdVideoStart : this.onAdVideoStart,
          AdVideoFirstQuartile : this.onAdVideoFirstQuartile,
          AdVideoMidpoint : this.onAdVideoMidpoint,
          AdVideoThirdQuartile : this.onAdVideoThirdQuartile,
          AdVideoComplete : this.onAdVideoComplete,
          AdUserAcceptInvitation : this.onAdUserAcceptInvitation,
          AdUserMinimize : this.onAdUserMinimize,
          AdUserClose : this.onAdUserClose,
          AdPaused : this.onAdPaused,
          AdPlaying : this.onAdPlaying,
          AdError : this.onAdError,
          AdLog : this.onAdLog
        };
        // Looping through the object and registering each of the callbacks with the
creative
        for ( var eventName in callbacks) {
          this._creative.subscribe(callbacks[eventName],
              eventName, this);
        }
      };

      // Pass through for initAd - when the video player wants to call the ad
      VPAIDWrapper.prototype.initAd = function(width, height,
          viewMode, desiredBitrate, creativeData,
          environmentVars) {
          this._creative.initAd(width, height, viewMode,
              desiredBitrate, creativeData,
              environmentVars);
      };
      // Callback for AdPaused
```

```
VPAIDWrapper.prototype.onAdPaused = function() {
  console.log("onAdPaused");
  .
  .
  .
};
// Callback for AdPlaying
VPAIDWrapper.prototype.onAdPlaying = function() {
  console.log("onAdPlaying");
  .
  .
  .
};
// Callback for AdError
VPAIDWrapper.prototype.onAdError = function(message) {
  console.log("onAdError: " + message);
  .
  .
  .
};
// Callback for AdLog
VPAIDWrapper.prototype.onAdLog = function(message) {
  console.log("onAdLog: " + message);
  .
  .
  .
};
// Callback for AdUserAcceptInvitation
VPAIDWrapper.prototype.onAdUserAcceptInvitation = function() {
  console.log("onAdUserAcceptInvitation");
  .
  .
  .
};
// Callback for AdUserMinimize
VPAIDWrapper.prototype.onAdUserMinimize = function() {
  console.log("onAdUserMinimize");
  .
  .
  .
};
// Callback for AdUserClose
VPAIDWrapper.prototype.onAdUserClose = function() {
  console.log("onAdUserClose");
  .
```

```
            .
            .
          };
         // Callback for AdUserClose
          VPAIDWrapper.prototype.onAdSkippableStateChange = function() {
            console.log("Ad Skippable State Changed to: " +
this._creative.getAdSkippableState());
            .
            .
            .
          };
          // Callback for AdUserClose
          VPAIDWrapper.prototype.onAdExpandedChange = function() {
            console.log("Ad Expanded Changed to: " + this._creative.getAdExpanded());
            .
            .
            .
          };
          // Pass through for getAdExpanded
          VPAIDWrapper.prototype.getAdExpanded = function() {
            console.log("getAdExpanded");
            return this._creative.getAdExpanded();
          };
          // Pass through for getAdSkippableState
          VPAIDWrapper.prototype.getAdSkippableState = function() {
            console.log("getAdSkippableState");
            return this._creative.getAdSkippableState();
          };
          // Callback for AdSizeChange
          VPAIDWrapper.prototype.onAdSizeChange = function() {
           console.log("Ad size changed to: w=" + this._creative.getAdWidth() + " h=" +
this._creative.getAdHeight());
            .
            .
            .
          };
          // Callback for AdDurationChange
          VPAIDWrapper.prototype.onAdDurationChange = function() {
            // console.log("Ad Duration Changed to: " + this._creative.getAdDuration());
            .
            .
            .
          };
          // Callback for AdRemainingTimeChange
          VPAIDWrapper.prototype.onAdRemainingTimeChange = function() {
                // console.log("Ad Remaining Time Changed to: " +
this._creative.getAdRemainingTime());
                .
                .
                .
            };
            // Pass through for getAdRemainingTime
            VPAIDWrapper.prototype.getAdRemainingTime = function() {
                console.log("getAdRemainingTime");
                return this._creative.getAdRemainingTime();
            };

            // Callback for AdImpression
            VPAIDWrapper.prototype.onAdImpression = function() {
                console.log("Ad Impression");
                .
                .
                .
            };
            // Callback for AdClickThru
            VPAIDWrapper.prototype.onAdClickThru = function(url, id, playerHandles) {
                console.log("Clickthrough portion of the ad was clicked");
                .
                .
                .
            };
            // Callback for AdInteraction
            VPAIDWrapper.prototype.onAdInteraction = function(id) {
                console.log("A non-clickthrough event has occured");
                .
                .
                .
            };


            // Callback for AdUserClose
            VPAIDWrapper.prototype.onAdVideoStart = function() {
                console.log("Video 0% completed");
                .
                .
                .
            };
            // Callback for AdUserClose
            VPAIDWrapper.prototype.onAdVideoFirstQuartile = function() {
                console.log("Video 25% completed");
```

```
  .
  .
  .
};
// Callback for AdUserClose
VPAIDWrapper.prototype.onAdVideoMidpoint = function() {
  console.log("Video 50% completed");
  .
  .
  .
};
// Callback for AdUserClose
VPAIDWrapper.prototype.onAdVideoThirdQuartile = function() {
  console.log("Video 75% completed");
  .
  .
  .
};
// Callback for AdVideoComplete
VPAIDWrapper.prototype.onAdVideoComplete = function() {
  console.log("Video 100% completed");
  .
  .
  .
};
// Callback for AdLinearChange
VPAIDWrapper.prototype.onAdLinearChange = function() {
  console.log("Ad linear has changed: " + this._creative.getAdLinear())
  .
  .
  .
};
// Pass through for getAdLinear
VPAIDWrapper.prototype.getAdLinear = function() {
  console.log("getAdLinear");
  return this._creative.getAdLinear();
};


// Pass through for startAd()
VPAIDWrapper.prototype.startAd = function() {
  console.log("startAd");
  this._creative.startAd();
};
// Callback for AdLoaded
```

```
VPAIDWrapper.prototype.onAdLoaded = function() {
  console.log("ad has been loaded");
  .
  .
  .
};
// Callback for StartAd()
VPAIDWrapper.prototype.onStartAd = function() {
  console.log("Ad has started");
  .
  .
  .
};
//Pass through for stopAd()
VPAIDWrapper.prototype.stopAd = function() {
  this._creative.stopAd();
};

// Callback for AdUserClose
VPAIDWrapper.prototype.onStopAd = function() {
  console.log("Ad has stopped");
  .
  .
  .
};

// Callback for AdUserClose
VPAIDWrapper.prototype.onSkipAd = function() {
  console.log("Ad was skipped");
  .
  .
  .
};
//Passthrough for setAdVolume
VPAIDWrapper.prototype.setAdVolume = function(val) {
  this._creative.setAdVolume(val);
};

//Passthrough for getAdVolume
VPAIDWrapper.prototype.getAdVolume = function() {
  return this._creative.getAdVolume();
};

// Callback for AdVolumeChange
VPAIDWrapper.prototype.onAdVolumeChange = function() {
  console.log("Ad Volume has changed to - " + this._creative.getAdVolume());
  .
  .
  .
};


//Passthrough for resizeAd
VPAIDWrapper.prototype.resizeAd = function(width, height,
    viewMode) {
  this._creative.resizeAd();
};
//Passthrough for pauseAd()
VPAIDWrapper.prototype.pauseAd = function() {
  this._creative.pauseAd();
};
//Passthrough for resumeAd()
VPAIDWrapper.prototype.resumeAd = function() {
  this._creative.resumeAd();
};
//Passthrough for expandAd()
VPAIDWrapper.prototype.expandAd = function() {
  this._creative.expandAd();
};
//Passthrough for collapseAd()
VPAIDWrapper.prototype.collapseAd = function() {
  this._creative.collapseAd();
};
```

9 Glossary

Video Ad – Advertisement that is displayed within the context of video content play. Note that the key part of the definition is that the content is video. The advertisement itself may or may not be a video.

Advanced Video Ad – Video advertisement with programming logic that enables user interactivity, interaction with the video content play, or other advanced features.

Companion Ad – Commonly text, display ad, rich media, or skin that wrap around the video experience. These ads come in a number of sizes and shapes, and typically run alongside or surrounding the video player.

Linear Video Ad – The ad is presented before, in the middle of, or after the video content is consumed by the user, in very much the same way a TV commercial can play before, during or after the chosen program.

Non-linear Video Ad – The ad unit runs concurrently with the video content so the users see the ad while viewing the content. Non-linear video ads can be delivered as text, graphical ads, or as video overlays.

Post-roll – a Linear Video ad spot that appears after the video content completes.

Pre-roll – a Linear Video ad spot that appears before the video content plays.

VAST (Video Ad Serving Template) – IAB-defined XML document format describing an ad unit to be displayed in, over, or around a Video Player.

Video Player – Environment in which in-stream video content is played. The Video Player may be built by the publisher or provided by a vendor.