



ATSC

ADVANCED TELEVISION
SYSTEMS COMMITTEE

ATSC Standard: ATSC 3.0 Interactive Content

A/344:2020
07 February 2020

Advanced Television Systems Committee
1776 K Street, N.W.
Washington, D.C. 20006
202-872-9160

The Advanced Television Systems Committee, Inc., is an international, non-profit organization developing voluntary standards and recommended practices for digital television. ATSC member organizations represent the broadcast, broadcast equipment, motion picture, consumer electronics, computer, cable, satellite, and semiconductor industries. ATSC also develops digital television implementation strategies and supports educational activities on ATSC standards. ATSC was formed in 1983 by the member organizations of the Joint Committee on Inter-society Coordination (JCIC): the Electronic Industries Association (EIA), the Institute of Electrical and Electronic Engineers (IEEE), the National Association of Broadcasters (NAB), the National Cable Telecommunications Association (NCTA), and the Society of Motion Picture and Television Engineers (SMPTE). For more information visit www.atsc.org.

Note: The user's attention is called to the possibility that compliance with this standard may require use of an invention covered by patent rights. By publication of this standard, no position is taken with respect to the validity of this claim or of any patent rights in connection therewith. One or more patent holders have, however, filed a statement regarding the terms on which such patent holder(s) may be willing to grant a license under these rights to individuals or entities desiring to obtain such a license. Details may be obtained from the ATSC Secretary and the patent holder.

Implementers with feedback, comments, or potential bug reports relating to this document may contact ATSC at <https://www.atsc.org/feedback/>.

Revision History

Version	Date
Candidate Standard approved	29 December 2016
A/344:2017 Standard approved	18 December 2018
Candidate Standard Revision approved See Annex C for details of WebSocket API changes	27 April 2018
CS update approved	24 January 2019
A/344:2019 approved	02 May 2019
A/344:2020 approved	07 January 2020

Table of Contents

1. SCOPE 1	
1.1 Introduction and Background	1
1.2 Organization	1
2. REFERENCES	2
2.1 Normative References	2
2.2 Informative References	3
3. DEFINITION OF TERMS	4
3.1 Compliance Notation	4
3.2 Treatment of Syntactic Elements	4
3.2.1 Reserved Elements	4
3.3 Acronyms and Abbreviations	5
3.4 Terms	5
4. OVERVIEW.....	7
4.1 Application Runtime Environment	7
4.2 Receiver Media Player Display	8
5. ATSC REFERENCE RECEIVER MODEL.....	10
5.1 Introduction	10
5.2 User Agent Definition	10
5.2.1 HTTP Protocols	10
5.2.2 XMLHttpRequest (XHR)	10
5.2.3 Cross-Origin Resource Sharing (CORS)	10
5.2.4 Mixed Content	11
5.2.5 Transparency	11
5.2.6 Full Screen	11
5.2.7 Visibility and Focus	11
5.3 Origin Considerations	11
6. BROADCASTER APPLICATION MANAGEMENT	14
6.1 Introduction	14
6.2 Application Context Cache Management	15
6.2.1 Signaling Intent for File Caching	15
6.2.2 Application Context Cache Hierarchy Definition	17
6.2.3 Active Service Application Context Cache Priority	18
6.2.4 Cache Expiration Time	19
6.2.5 Advanced Emergency Alert Enhancement Content Considerations	19
6.3 Broadcaster Application Signaling	20
6.3.1 Broadcaster Application Launch	20
6.3.2 Broadcaster Application Events (Static / Dynamic)	20
6.4 Broadcaster Application Delivery	21
6.4.1 Broadcaster Application Packages	21
6.4.2 Broadcaster Application Package Changes	22
6.5 Security Considerations	22
6.6 Companion Device Interactions	22

7. MEDIA PLAYER.....	23
7.1 Utilizing RMP	23
7.1.1 Broadcast or Hybrid Broadband and Broadcast Live Streaming	23
7.1.2 Broadband Media Streaming	23
7.1.3 Downloaded Media Content	24
7.1.4 Redistribution	24
7.2 Utilizing AMP	24
7.2.1 Broadcast or Hybrid Broadband and Broadcast Live Streaming	24
7.2.2 Broadband Media Streaming	24
7.2.3 Downloaded Media Content	24
7.2.4 AMP Utilizing the Pull Model	24
7.2.5 AMP Utilizing the Push Model	24
8. ATSC 3.0 WEB SOCKET INTERFACE.....	25
8.1 Introduction	25
8.2 Interface Binding	26
8.2.1 WebSocket Servers	26
8.3 Data Binding	28
8.3.1 Cancel Request Command	29
8.3.2 Error Handling	33
9. SUPPORTED METHODS.....	34
9.1 API Revision Control	34
9.2 Receiver Query APIs	36
9.2.1 Query Content Advisory Rating API	36
9.2.2 Query Closed Captions Enabled/Disabled API	38
9.2.3 Query Service ID API	39
9.2.4 Query Language Preferences API	40
9.2.5 Query Caption Display Preferences API	41
9.2.6 Query Audio Accessibility Preferences API	43
9.2.7 Query Receiver Web Server URI API	45
9.2.8 Query Alerting Signaling API	46
9.2.9 Query Service Guide URLs API	48
9.2.10 Query Signaling Data API	50
9.3 Asynchronous Notifications of Changes	52
9.3.1 Content Advisory Rating Change Notification API	53
9.3.2 Content Advisory Rating Block Change Notification API	54
9.3.3 Service Change Notification API	55
9.3.4 Caption State Change Notification API	56
9.3.5 Language Preference Change Notification API	56
9.3.6 Caption Display Preferences Change Notification API	57
9.3.7 Audio Accessibility Preference Change Notification API	60
9.3.8 Alerting Change Notification API	62
9.3.9 Content Change Notification API	64
9.3.10 Service Guide Change Notification API	65
9.3.11 Signaling Data Change Notification API	67
9.4 Cache Request APIs	68
9.4.1 Cache Request API	68
9.4.2 Cache Request DASH API	71

9.5	Query Cache Usage API	76
9.6	Event Stream APIs	77
9.6.1	Event Stream Subscribe API	77
9.6.2	Event Stream Unsubscribe API	79
9.6.3	Event Stream Event API	80
9.7	Request Receiver Actions	82
9.7.1	Acquire Service API	82
9.7.2	Video Scaling and Positioning API	83
9.7.3	Set RMP URL API	85
9.7.4	Audio Volume API	91
9.7.5	Integrated Subscribe / Unsubscribe API for Notifications	93
9.8	Media Track Selection API	97
9.9	Mark Unused API	98
9.10	Content Recovery APIs	100
9.10.1	Query Content Recovery State API	100
9.10.2	Query Display Override API	102
9.10.3	Query Recovered Component Info API	104
9.10.4	Content Recovery State Change Notification API	105
9.10.5	Display Override Change Notification API	107
9.10.6	Recovered Component Info Change Notification API	107
9.11	Filter Codes APIs	108
9.11.1	Get Filter Codes API	109
9.11.2	Set Filter Codes API	110
9.12	Keys APIs	111
9.12.1	Request Keys API	111
9.12.2	Relinquish Keys API	113
9.12.3	Request Keys Timeout	114
9.13	Query Device Info API	115
9.14	RMP Content Synchronization APIs	119
9.14.1	Query RMP Media Time API	119
9.14.2	Query RMP Wall Clock API	120
9.14.3	Query RMP Playback State API	121
9.14.4	Query RMP Playback Rate API	122
9.14.5	RMP Media Time Change Notification API	123
9.14.6	RMP Playback State Change Notification API	124
9.14.7	RMP Playback Rate Change Notification API	124
9.15	DRM APIs	125
9.15.1	DRM Notification API	125
9.15.2	DRM Operation API	127
9.16	XLink APIs	129
9.16.1	XLink Resolution Notification API	129
9.16.2	XLink Resolved API	130
ANNEX A	DASH AD INSERTION.....	136
A.1	Introduction	136
A.2	Dynamic Ad Insertion Principles	136
A.3	Overview of XLinks	137

ANNEX B	OBSCURING THE LOCATION OF AD AVAILS	139
B.1	Obscuring the Location of Ad Avails	139
ANNEX C	A/344:2019 WS API CHANGES	142
C.1	Introduction	142
C.2	Query Alerting URL and Alerting Change Notification APIs	142
C.2.1	Query Alerting URL API [Deprecated]	142
C.2.2	Alerting Change Notification API [Deprecated]	144
C.3	Extended Video Scaling and Positioning API	146
C.4	Extended Set RMP URL API	147
C.5	Added APIs	147
C.6	Deprecated APIs	147
C.6.1	XLink Resolution API [Deprecated]	147
C.6.2	Subscribe MPD Changes API [Deprecated]	149
C.6.3	Unsubscribe MPD Changes API [Deprecated]	149
C.6.4	Subscribe Alerting Changes API [Deprecated]	150
C.6.5	Unsubscribe Alerting Changes API [Deprecated]	151
C.6.6	Subscribe Content Changes API [Deprecated]	152
C.6.7	Unsubscribe Content Changes API [Deprecated]	153
C.6.8	Subscribe RMP Media Time Change Notification API [Deprecated]	153
C.6.9	Unsubscribe RMP Media Time Change Notification API [Deprecated]	154
C.6.10	Subscribe RMP Playback State Change Notification API [Deprecated]	155
C.6.11	Unsubscribe RMP Playback State Change Notification API [Deprecated]	155
C.6.12	Subscribe RMP Playback Rate Change Notification API [Deprecated]	156
C.6.13	Unsubscribe RMP Playback Rate Change Notification API [Deprecated]	157
ANNEX D	A/344:2020 WS API CHANGES	158
D.1	Introduction	158
D.2	Added APIs	158
D.3	Deprecated APIs	158
D.3.1	Query MPD URL API [Deprecated]	158
D.3.2	MPD Change Notification API [Deprecated]	159
ANNEX E	JSON-RPC 2.0 SPECIFICATION	160
1	Overview	160
2	Conventions	160
3	Compatibility	161
4	Request object	161
5	Response object	162
6	Batch	163
7	Examples	163
8	Extensions	166

Index of Figures and Tables

Figure 4.1 Rendering model for application enhancements using RMP.	9
Figure 5.1 ATSC 3.0 Reference Receiver Model Logical Components.	10
Figure 5.2 Application Context Identifier Conceptual Model.	13
Figure 6.1 Receiver Conceptual Architecture.	14
Figure 6.2 Example Application Context Cache Hierachy.	18
Figure 8.1 Communication with ATSC 3.0 receiver.	25
Figure 9.1 RMP audio volume.	92
Figure A.1 Personalized ad periods.	137
Figure A.2 Example period with XLink.	137
Figure A.3 Example remote period.	138
Figure B.1 Public and private MPDs.	139
Figure B.2 Example public MPD.	139
Figure B.3 Example MPD equivalent.	140
Figure B.4 Example ad replacement.	141
Table 4.1 Application Actions and APIs	8
Table 6.1 ATSC-Defined Extension to the <code>metadataEnvelope.item</code> Element	16
Table 8.1 WebSocket Server Functions and URLs	27
Table 8.2 JSON-RPC ATSC Error Codes	33
Table 9.1 API Applicability	35
Table 9.2 Asynchronous Notifications	52
Table 9.3 Subscription Parameter List	93
Table 9.4 Input Key Code Values	117

ATSC Standard: ATSC 3.0 Interactive Content

1. SCOPE

This document describes the interactive content environment provided by an ATSC 3.0 receiver. This environment is comprised of a standard W3C User Agent with known characteristics, a WebSocket interface for obtaining information from the receiver and controlling various receiver functionality, and an HTTP interface for accessing files delivered over broadcast. This document also specifies the life cycle of the interactive content when delivered over broadband or broadcast or both.

ATSC 3.0 is a defining emission standard, while the W3C User Agent defines a standard environment for executing interactive content. In order to create the appropriate user experience that aligns with the ATSC 3.0 delivery mechanisms, it is considered useful to specify a reference architecture of an ATSC 3.0 receiver device, referred to in this document as the Reference Receiver Model (RRM) or simply the Receiver, to define and/or verify the proper emission specifications and to allow interactive content developers to target a known environment (see Section 5).

A decomposition of the functions and interfaces in the Receiver enables the definition of proper emission formats in order to verify that the distribution formats result in expected functionality to fulfill the ATSC 3.0 system requirements.

By no means would such a reference receiver imply a normative implementation, as it would only provide an example implementation to verify the adequacy of the delivery specification. The RRM is expected to decompose the ATSC 3.0 receiver device into the relevant network interfaces, device internal functions, interfaces to the Broadcaster Application and interfaces to the media playout pipeline.

It should be noted that the word “shall” is used to describe how an interface or method is expected to work. It is anticipated that if the Receiver implements the interface or method, that the resultant behavior is consistent with this specification. This allows interactive content developers to implement to a well-defined application programming interface.

1.1 Introduction and Background

This document describes the environment and interfaces that can be used by interactive content to provide an enhanced viewer experience on a supporting ATSC 3.0 receiver.

1.2 Organization

This document is organized as follows:

- Section 1 – The scope, introduction, and background of this specification
- Section 2 – Normative and informative references
- Section 3 – Compliance notation, definition of terms, and acronyms
- Section 4 – Overview of the interactive content environment from the system level
- Section 5 – Specification of the Reference Receiver Model
- Section 6 – Describes how the Broadcaster Application is managed
- Section 7 – Details of the various Media Players supported by this standard
- Section 8 – Overview of the WebSocket interface supported by the receiver
- Section 9 – Supported methods of the WebSocket interface

- Annex A – Principles of DASH ad insertion
- Annex B – Practical discussion of how to obscure advertising avails
- Annex C – A/344:2019 WS API Changes
- Annex D – A/344:2020 WS API Changes
- Annex E – A complete copy of the JSON RPC 2.0 specification used by this standard

2. REFERENCES

All referenced documents are subject to revision. Users of this Standard are cautioned that newer editions might or might not be compatible.

2.1 Normative References

The following documents, in whole or in part, as referenced in this document, contain specific provisions that are to be followed strictly in order to implement a provision of this Standard.

- [1] ATSC: “ATSC Standard: Signaling, Delivery, Synchronization, and Error Protection (A/331),” Doc. A/331:2019, Advanced Television Systems Committee, 20 June 2019.
- [2] ATSC: “ATSC Standard: Service Announcement,” Doc. A/332:2017, Advanced Television Systems Committee, 6 December 2017.
- [3] ATSC: “ATSC Standard: Content Recovery in Redistribution Scenarios,” Doc. A/336:2019, Advanced Television Systems Committee, 3 October 2019.
- [4] ATSC: “ATSC Standard: Application Event Delivery (A/337),” Doc. A/337:2019, Advanced Television Systems Committee, 30 April 2019.
- [5] ATSC: “ATSC Standard: Captions and Subtitles (A/343),” Doc. A/343:2018, Advanced Television Systems Committee, 10 October 2018.
- [6] ATSC: “ATSC Standard: ATSC 3.0 Security and Service Protection,” Doc. A/360:2019, Advanced Television Systems Committee, 20 August 2019.
- [7] CTA: “CTA Specification: Web Application Video Ecosystem – Web Media API Snapshot 2017”, Doc. CTA-5000, Consumer Technology Association, December 2017.
- [8] IEEE: “Use of the International Systems of Units (SI): The Modern Metric System,” Doc. SI 10, Institute of Electrical and Electronics Engineers, New York, N.Y.
- [9] IETF: “Augmented BNF for Syntax Specifications: ABNF,” RFC 5234, Internet Engineering Task Force, January 2008. <https://tools.ietf.org/html/rfc5234>
- [10] IETF: “Hypertext Transfer Protocol (HTTP/1.1): Authentication,” Doc. RFC 7235, Internet Engineering Task Force, June 2014. <https://tools.ietf.org/html/rfc7235>
- [11] IETF: “Hypertext Transfer Protocol (HTTP/1.1): Caching,” Doc. RFC 7234, Internet Engineering Task Force, June 2014. <https://tools.ietf.org/html/rfc7234>
- [12] IETF: “Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests,” Doc. RFC 7232, Internet Engineering Task Force, June 2014. <https://tools.ietf.org/html/rfc7232>
- [13] IETF: “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” Doc. RFC 7230, Internet Engineering Task Force, June 2014. <https://tools.ietf.org/html/rfc7230>

- [14] IETF: “Hypertext Transfer Protocol (HTTP/1.1): Range Requests,” Doc. RFC 7233, Internet Engineering Task Force, June 2014.
<https://tools.ietf.org/html/rfc7233>
- [15] IETF: “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” Doc. RFC 7231, Internet Engineering Task Force, June 2014.
<https://tools.ietf.org/html/rfc7231>
- [16] IETF: “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,” RFC 2045, Internet Engineering Task Force, November 1996.
<https://tools.ietf.org/html/rfc2045>
- [17] IETF: “Tags for Identifying Languages,” RFC 5646, Internet Engineering Task Force, September 2009. <https://tools.ietf.org/html/rfc5646>
- [18] IETF: “The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 7159, Internet Engineering Task Force, March 2014. <https://tools.ietf.org/html/rfc7159>
- [19] IETF: “The Web Origin Concept,” RFC 6454, Internet Engineering Task Force, December 2011. <https://tools.ietf.org/html/rfc6454>
- [20] IETF: “The WebSocket Protocol,” RFC 6455, Internet Engineering Task Force, December 2011. <https://tools.ietf.org/html/rfc6455>
- [21] IETF: “Uniform Resource Identifier (URI): Generic Syntax,” RFC 3986, Internet Engineering Task Force, January 2005. <https://tools.ietf.org/html/rfc3986>
- [22] IETF: “A Universally Unique IDentifier (UUID) URN Namespace,” Doc. RFC 4122, Internet Engineering Task Force, July 2005. <https://tools.ietf.org/html/rfc4122>
- [23] ISO/IEC: ISO/IEC 23009-1:2014, “Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats,” International Organization for Standardization, 15 May 2014.
- [24] W3C: “Encrypted Media Extensions,” W3C Recommendation, World Wide Web Consortium, 18 September 2017. <http://www.w3.org/TR/encrypted-media/>
- [25] W3C: “UI Events KeyboardEvent key Values,” Section 3.18, Media Controller Keys, W3C Candidate Recommendation, 1 June 2017, World Wide Web Consortium.
<https://www.w3.org/TR/DOM-Level-3-Events-key/#keys-media-controller>
- [26] W3C: “Media Source Extensions,” W3C Recommendation, World Wide Web Consortium, 17 November 2016. <https://www.w3.org/TR/media-source/>
- [27] W3C: “Mixed Content,” W3C Candidate Recommendation, Worldwide Web Consortium, 2 August 2016. (*work in process*) <http://www.w3.org/TR/mixed-content/>
- [28] W3C: “XML Schema Part 2: Datatypes Second Edition” W3C Recommendation, Worldwide Web Consortium, 28 October 2004. <https://www.w3.org/TR/xmlschema-2/>

2.2 Informative References

The following documents contain information that may be helpful in applying this Standard.

- [29] ATSC: “ATSC Standard: Companion Device,” Doc. A/338:2019, Advanced Television Systems Committee, 2 October 2019.
- [30] ATSC: “ATSC Recommended Practice: Techniques for Signaling, Delivery and Synchronization,” Doc. A/351:2019, Advanced Television Systems Committee, 28 August 2019.
- [31] CTA: “Digital Television (DTV) Closed Captioning,” Doc. CTA-708, Consumer Technology Association, Arlington, VA.

- [32] DASH-IF: “Guidelines for Implementation: DASH-IF Interoperability Point for ATSC 3.0,” Version 1.0, DASH Industry Forum, 3 May 2017.
- [33] IEEE: IEEE Registration Authority.
<https://regauth.standards.ieee.org/standards-ra-web/pub/view.html>
- [34] JSON-RPC: “JSON-RPC 2.0 Specification,” JSON-RPC Working Group.
<http://www.jsonrpc.org/specification>
- [35] JSON Schema: “JSON Schema: A Media Type for Describing JSON Documents,” Internet Engineering Task Force, JSON-Schema Working Group, 12 April 2017.
<http://json-schema.org/latest/json-schema-core.html> (*work in progress*)
- [36] W3C: “TTML Profiles for Internet Media Subtitles and Captions 1.0 (IMSC1),” W3C Recommendation, Worldwide Web Consortium. <http://www.w3.org/TR/ttml-imsc1>
- [37] W3C: “XML Linking Language (XLink),” Recommendation Version 1.1, Worldwide Web Consortium, 6 May 2010. <http://www.w3.org/TR/xlink11/>
- [38] WHATWG: “HTML Living Standard,” Section 9.3 “Web sockets,” Web Hypertext Application Technology Working Group.
<https://html.spec.whatwg.org/multipage/web-sockets.html>

3. DEFINITION OF TERMS

With respect to definition of terms, abbreviations, and units, the practice of the Institute of Electrical and Electronics Engineers (IEEE) as outlined in the Institute’s published standards [8] shall be used. Where an abbreviation is not covered by IEEE practice or industry practice differs from IEEE practice, the abbreviation in question is described in Section 3.3 of this document.

3.1 Compliance Notation

This section defines compliance terms for use by this document:

shall – This word indicates specific provisions that are to be followed strictly (no deviation is permitted).

shall not – This phrase indicates specific provisions that are absolutely prohibited.

should – This word indicates that a certain course of action is preferred but not necessarily required.

should not – This phrase means a certain possibility or course of action is undesirable but not prohibited.

3.2 Treatment of Syntactic Elements

This document contains symbolic references to syntactic elements used in the audio, video, and transport coding subsystems. These references are typographically distinguished by the use of a different font (e.g., *restricted*), may contain the underscore character (e.g., *sequence_end_code*) and may consist of character strings that are not English words (e.g., *dynrng*).

3.2.1 Reserved Elements

One or more reserved bits, symbols, fields, or ranges of values (i.e., elements) may be present in this document. These are used primarily to enable adding new values to a syntactical structure without altering its syntax or causing a problem with backwards compatibility, but they also can be used for other reasons.

The ATSC default value for reserved bits is ‘1’. There is no default value for other reserved elements. Use of reserved elements except as defined in ATSC Standards or by an industry

standards setting body is not permitted. See individual element semantics for mandatory settings and any additional use constraints. As currently reserved elements may be assigned values and meanings in future versions of this Standard, receiving devices built to this version are expected to ignore all values appearing in currently reserved elements to avoid possible future failure to function as intended.

3.3 Acronyms and Abbreviations

The following acronyms and abbreviations are used within this document.

AEAT	Advanced Emergency Alert Table [1]
AMP	Application Media Player
API	Application Programming Interface
ATSC	Advanced Television Systems Committee
CSS	Cascading Style Sheets
DASH	Dynamic Adaptive Streaming over HTTP [23]
EME	W3C Encrypted Media Extensions [24]
ESG	Electronic Service Guide [2]
HELD	HTML Entry pages Location Description [1]
HTML5	HyperText Markup Language, Fifth Version
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation [18]
MPD	Media Presentation Description [32]
MSE	W3C Media Source Extensions [26]
OSN	On Screen message Notification [1]
RMP	Receiver Media Player
ROUTE	Real-Time Object Delivery over Unidirectional Transport [1]
RPC	Remote Procedure Call
RRM	Reference Receiver Model
URL	Uniform Resource Locator
W3C	Worldwide Web Consortium
XML	eXtensible Markup Language

3.4 Terms

The following terms are used within this document.

Application Context Cache – The Application Context Cache is a conceptual storage area where information from the broadcast is collected for retrieval through the Receiver Web Server. This document refers to the Application Context Cache as if it were implemented as actual storage though this is for convenience only. An Application Context Cache corresponds to the Application Context Identifier associated with each Broadcaster Application. Files delivered over ROUTE contain attributes that determine the Application Context Cache with which they will be associated.

Application Context Identifier – An Application Context Identifier is a unique URI that determines which resources are provided to an associated Broadcaster Application by the Receiver. Resources may be associated with multiple Application Context Identifiers, but a

Broadcaster Application is only associated with a single Application Context Identifier. Details of the Application Context Identifier syntax are specified in the HELD [1].

Base URI – The Base URI specifies the initial portion of a URL used by the Broadcaster Application to access files within the Application Context Cache. The Base URI is prepended to the relative URI path of a file to obtain the full URL of the file within the Application Context Cache. The Base URI is uniquely generated by the Receiver based on the Application Context Identifier defined for the Broadcaster Application.

Broadcaster Application – A Broadcaster Application is used herein to refer to the functionality embodied in a collection of files comprised of an HTML5 document, known as the Entry Page and other HTML5, CSS, JavaScript, image and multimedia resources referenced directly or indirectly by that document, all provided by a broadcaster in an ATSC 3.0 service. The Broadcaster Application refers to the client-side functionality of the broader Web Application that provides the interactive service. The distinction is made because the broadcaster only transmits the client-side documents and code. The server-side of this broader Web Application is implemented by an ATSC 3.0 receiver and has a standardized API for all applications. No server-side application code can be supplied by the broadcaster. The broadcaster may provide Web-based documents and code that work in conjunction with the Broadcaster Application over broadband making the Broadcaster Application a true Web Application. The collection of files making up the Broadcaster Application can be delivered over the web in a standard way or can be delivered over broadcast as packages via the ROUTE protocol.

Entry Package – The Entry Package contains one or more files that comprise the functionality of the Broadcaster Application. The Entry Package includes the Entry Page and perhaps additional supporting files including JavaScript, CSS, image files and other content.

Entry Page – The Entry Page is the initial HTML5 document referenced by application signaling that should be loaded first into the User Agent. The Entry Page is one of the files in the Entry Package.

Event Stream – An Event Stream is a series of messages, either static, in DASH signaling, or dynamic, contained in defined messages within media segments. The events contained within the Event Stream can initiate interactive actions on the part of a Broadcaster Application.

Receiver – The Receiver described in this document refers to an entity that implements the functions of the Reference Receiver Model.

Receiver Web Server – The Receiver Web Server is a conceptual component of a Receiver that provides a means for a User Agent to gain access to files delivered over ROUTE that conceptually reside in the Application Context Cache.

Receiver WebSocket Server – The Receiver WebSocket Server provides a means for a User Agent to gain access to information about the Receiver and control various features provided by the Receiver.

Redistribution – A use case wherein an ATSC 3.0 service is delivered to a Receiver via a protocol other than ATSC 3.0; e.g. HDMI.

Reference Receiver Model – A conceptual receiver device that is capable of executing the APIs and behavior specified in this document. This document specifies normative attributes of the model, which are intended to inform actual receiver implementations.

reserved – Set aside for future use by a Standard.

User Agent – Software provided by the Receiver that retrieves and renders Web content. The User Agent interprets HTML5, CSS, and JavaScript, renders media, text, and graphics, and can create user interaction dialogs.

Web Application – A Web Application is a client/server program accessed via the web using URLs. The client-side software is executed by a User Agent.

4. OVERVIEW

4.1 Application Runtime Environment

This specification defines the details of an environment that is required for a Broadcaster Application to run. In the broadcast environment, the files associated with a Broadcaster Application are delivered in ROUTE packages that are unpacked into a conceptual cache area. The pages and resources of a Broadcaster Application are then made available to the User Agent associated with the Receiver. In the broadband environment, launching a Broadcaster Application behaves in the same way as in a normal web environment with no specialized behavior or intervention from the Receiver.

The Broadcaster Application executes inside a W3C-compliant User Agent accessing some of the graphical elements of the Receiver to render the user interface or accessing some of the resources or information provided by the Receiver. If a Broadcaster Application requires access to resources such as information known to the Receiver, or if the Broadcaster Application requires the Receiver to perform a specific action that is not defined by standard W3C User Agent APIs that are widely implemented by browsers, then the Broadcaster Application sends a request to the Receiver WebSocket Server utilizing the set of JSON-RPC messages defined in this specification.

The JSON-RPC messages defined in this specification provide the APIs that are required by the Broadcaster Application to access the resources that are otherwise not reachable. These JSON-RPC messages allow the Broadcaster Application to query information that is gathered or collected in the Receiver, to receive notifications via broadcast signaling, and to request performing of actions that are not otherwise available via the standard JavaScript APIs.

There are noteworthy differences between an HTML5 application deployed in a normal web environment and one deployed in an ATSC 3.0 broadcast environment. In the ATSC 3.0 broadcast environment, a Broadcaster Application can:

- Access resources from broadcast or broadband;
- Request Receivers to perform certain functions that are not otherwise available via the JavaScript APIs, such as:
 - Utilizing the media player provided by the Receiver (called the Receiver Media Player) to:
 - Stream media content via broadcast signaling delivery mechanism
 - Stream media content (i.e. unicast) via broadband delivery mechanism
 - Playback media content that has been downloaded via broadcast or broadband delivery mechanisms;
 - Utilizing MSE and EME to play media content streamed over broadcast or broadband
- Query information that is specific to the reception of TV services, for example, the status of closed caption display and language references, and receive notifications of changes in this information;

- Receive notifications of “stream events” that are embedded in the media content or signaling, when that media content is being played by the Receiver Media Player.

Another noteworthy difference between the two models is that in the normal web environment, the viewer is in direct control of launching an HTML5 application by specifying the URL of a desired website. In the ATSC 3.0 environment, although the user still initiates the action by selecting a Service, the actual application URL is not explicitly selected by the viewer and instead is provided via broadcast signaling. In this case, it is the responsibility of the Receiver using its User Agent to launch or terminate the Broadcaster Application referenced by a URL provided in broadcast signaling.

The Broadcaster Application relies on a set of features that are provided via the User Agent. Although it is beyond the scope of this specification to describe how the pages of a Broadcaster Application are provided to the User Agent, it is recommended that standard web technologies should be used to serve the pages.

Table 4.1 shows which type of API a broadcaster-provided application uses to access the features provided by the Receiver.

Table 4.1 Application Actions and APIs

Action Requested by the Application	API Used by the Application
Requesting to download a media file from broadband	W3C APIs provided via the user-agent
Querying information related to user display and presentation preferences, including languages, accessibility options, and closed caption settings	Receiver WebSocket Server APIs, described in this specification in Section 9.2
Requesting to stream downloaded media file from broadcast	Via push or pull model, described in this specification in Sections 9.2 and 9.6.2
Requesting to stream downloaded media file from broadband	Via push or pull model, described in this specification in Sections 9.2 and 9.6.2
Requesting the Receiver Media Player to play a broadband-delivered media stream	Receiver WebSocket Server APIs, described in this specification in Section 9.7.3
Subscribing (or un-subscribing) to stream event notifications that are sent as part of ROUTE/DASH over broadcast	Receiver WebSocket Server APIs, described in this specification in Sections 9.6.1 and 9.6.2
Receiving stream event notifications that are sent as part of ROUTE/DASH over broadcast	Receiver WebSocket Server APIs, described in this specification in Section 9.6.3
Querying the Receiver to learn the identity of the currently-selected broadcast service	Receiver WebSocket Server APIs, described in this specification in Section 9.2.3
Receiving notice of changes to user display and presentation preferences	Receiver WebSocket Server APIs, described in this specification in Section 9.3.6
Requesting the Receiver to select a new broadcast service	Receiver WebSocket Server APIs, described in this specification in Section 9.7.1

4.2 Receiver Media Player Display

The RMP presents its video output behind any visible output from the Broadcaster Application. Figure 4.1 illustrates the relationship and the composition function performed in the Receiver.

Figure 4.1 illustrates two examples. In the example on the left, the graphical output from the Broadcaster Application is overlaid onto the full-screen video being rendered by the Receiver Media Player. For the linear A/V service with application enhancement, the Broadcaster Application may instruct the Receiver Media Player to scale the video, as it may wish to use more area for graphics. A JSON-RPC message as described in Section 9.7.2 is used to instruct the RMP to scale and position the video it renders. This scenario is illustrated in the example shown on the

right side of the figure. The Broadcaster Application will likely want to define the appearance of the screen surrounding the video inset. It can do that by defining the background in such a way that the rectangular area where the RMP video is placed is specified as transparent.

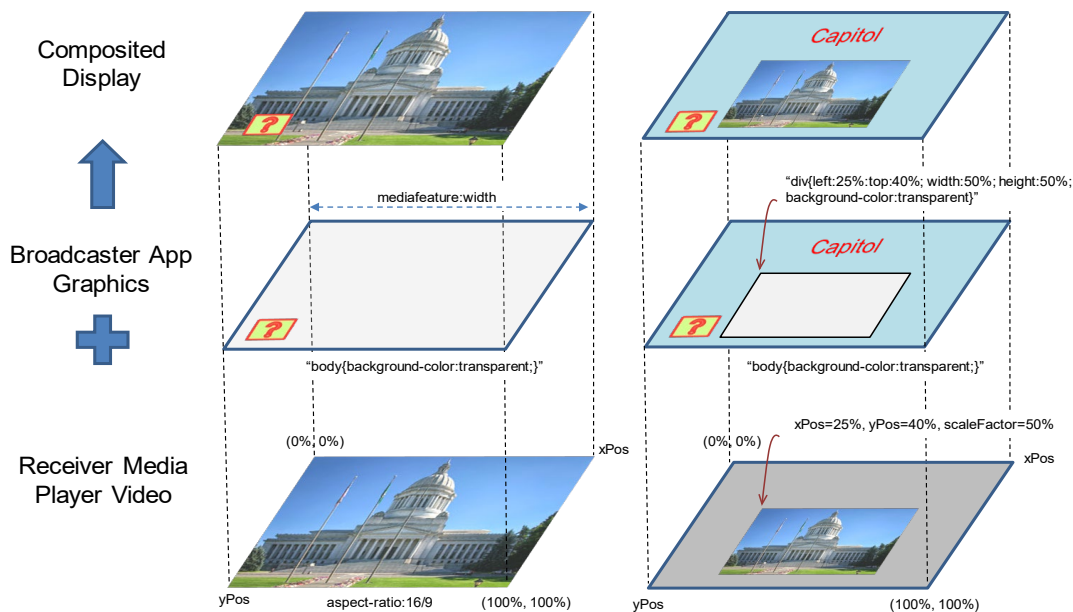


Figure 4.1 Rendering model for application enhancements using RMP.

A Broadcaster Application can expect that the User Agent graphics window, [0,0] to a full 100% in both axes, maps directly to the RMP logical video display window at its full dimensions. Since most Receiver user interfaces may not conveniently enable scroll bar manipulation, the Broadcaster Application should consider disabling scroll bars using standard W3C mechanisms in most situations. Note that the display of closed captioning is directly related to the current audio selected and is expected to be presented on top of all other content and video. However, the display of closed captioning is the responsibility of the Receiver.

A Receiver may choose to render its own native application on top of the Broadcaster Application due to some user interaction or other similar events. For example, this may happen when the viewer chooses to configure the Receiver settings while a Broadcaster Application is active.

When the Receiver presents its own native application, the Receiver, through standard W3C notification methods, shall notify the Broadcaster Application that it no longer has the focus. The Broadcaster Application may choose to either hide itself or maintain its current display. This behavior is left up to design of each Broadcaster Application.

Additionally, the Receiver may choose to hide the launched Broadcaster Application to avoid issues with scaling video and a full-scale Broadcaster Application. The behavior of whether the Broadcaster Application is hidden or not is left up to the Receiver, but the Receiver shall not terminate the Broadcaster Application, as long as the associated service remains selected and application signaling has not selected another Broadcaster Application.

Regardless of whether the Broadcaster Application is hidden or behind a Receiver native application, the Broadcaster Application is notified that it has lost its focus via standard W3C notification methods.

5. ATSC REFERENCE RECEIVER MODEL

5.1 Introduction

An ATSC 3.0 Reference Receiver Model may be composed of several logical components, which are described in this section. In practice, several of the given logical components can be combined into one component or one logical component can be divided into multiple components. Figure 5.1 shows the logical components of an ATSC 3.0 Reference Receiver Model. Although the software stack shows a layering architecture, it does not necessarily mean one module must use the layer below to access other modules in the system, with the exception of the Broadcaster Applications, which are run in the User Agent implementation provided by the Receiver, which complies with the APIs specified in this specification.

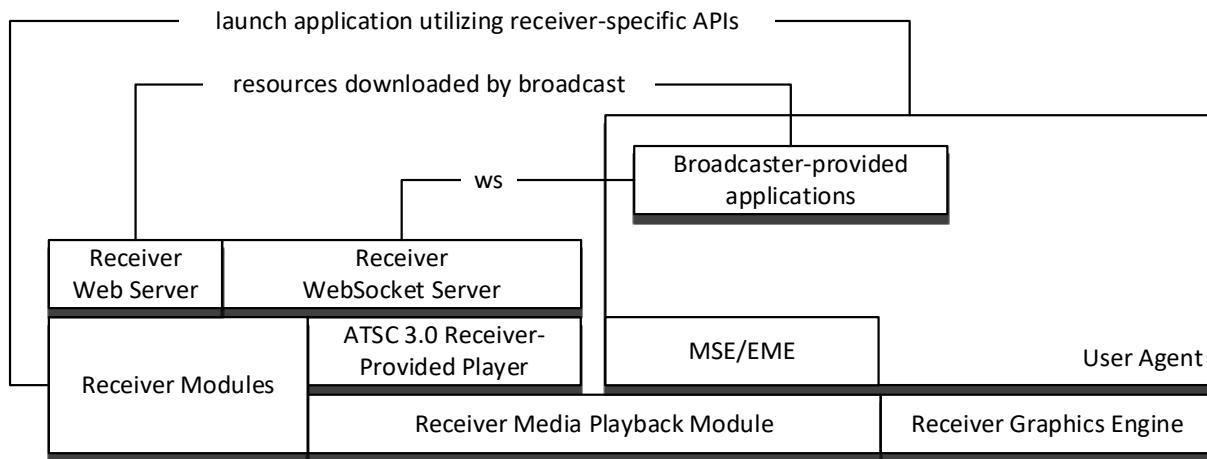


Figure 5.1 ATSC 3.0 Reference Receiver Model Logical Components.

5.2 User Agent Definition

Receivers shall implement an HTML5 User Agent that complies with all normative requirements specified in the CTA Web Media API Snapshot (CTA-5000) [7]. In addition, the features described in the following sections shall be supported.

5.2.1 HTTP Protocols

The User Agent shall implement the HTTP protocols specified in RFC 7230 through RFC 7235, references [10], [11], [12], [13], [14] and [15]. User Agents shall implement the Web Origin Concept specification [19] and the HTTP State Management Mechanism specification (Cookies) ([7] Section 4.2) as well. These are referenced in [7] as [HTTP], [ORIGIN], and [COOKIES].

5.2.2 XMLHttpRequest (XHR)

The User Agent shall support the XMLHttpRequest and related interfaces of the [XHR] reference in [7]. In the case of an XHR request where the request URL identifies a broadcast resource, the request is delivered to the Receiver Web Server, rather than to an Internet web server.

5.2.3 Cross-Origin Resource Sharing (CORS)

The User Agent shall support the [ORIGIN] specification referenced in [19].

5.2.4 Mixed Content

The User Agent shall handle fetching of content over unencrypted or unauthenticated connections in the context of an encrypted and authenticated document according to the W3C Mixed Content specification [27] though Broadcaster Applications are encouraged to only reference trusted content. References to files within the Application Context Cache (see Section 5.3 below) shall be considered to be “a priori authenticated” in the terminology of W3C Mixed Content. Any resource accessed from the Application Context Cache shall be considered to have been accessed within a secure context.

5.2.5 Transparency

The background of the User Agent’s drawing window might be transparent by default. Nevertheless, it is recommended that Broadcaster Applications explicitly specify the areas desired to be opaque or transparent to maintain consistency across Receivers. Thus, for example, if any element in the web page (such as a table cell) includes a CSS style attribute “background-color: transparent”, and that area is not covered by another layer with an opaque element, then video content presented by the Receiver Media Player (see Section 4.2) might be visible in that area. Note that certain areas can be specified as transparent while others are opaque.

5.2.6 Full Screen

As stated in Section 4.2, a Broadcaster Application can expect that the User Agent graphics window, [0,0] to a full 100% in both axes, maps directly to the RMP logical video display window at its full dimensions. The “width” media feature of CSS MediaQuery [7] shall align with the width of the RMP logical video display window. In most viewing conditions, the RMP logical video display window will fill the entire screen.

5.2.7 Visibility and Focus

The Receiver shall use the W3C Page Visibility Level 2 API as required by CTA-5000 [7] to inform the Broadcaster Application whether its display output is visible or not. The Receiver may choose to obscure or mute the Broadcaster Application display output for a variety of reasons including but not limited to display of Receiver preference dialogs, content blocking, or emergency information display. Similarly, the Receiver shall provide the W3C Focus APIs as required by CTA-5000 [7] to allow the Broadcaster Application to determine if it can receive user input or not.

5.3 Origin Considerations

Each file that is delivered via broadband has the usual absolute URL associated with it. Each file that is delivered via broadcast has a relative URL reference associated with it, signaled in the broadcast, and it also has one or more Application Context Identifiers associated with it, signaled in the broadcast. As specified below, Receivers assign to each broadcast file a Base URI that converts the relative URL reference to one or more absolute URLs, taking its Application Context Identifier(s) into account.

An Application Context Identifier is a unique URI that determines which resources are provided to an associated Broadcaster Application by the Receiver. The Application Context Identifier to be bound to the Broadcaster Application is signaled in the HELD [4]. An Application Context Identifier may be associated with many Broadcaster Applications; however, each Broadcaster Application shall be associated with a single Application Context Identifier. Each Application Context Identifier forms a unique conceptual environment in which the Receiver is expected to comingle resources for use by the associated Broadcaster Applications. This unique conceptual environment is referred to herein as the Application Context Cache.

The origin of a web resource is defined in RFC 6454 [19]. While the technical description in RFC 6454 is convoluted to cover the multitude of edge cases, the resultant URIs should be familiar with a “scheme” portion (e.g., “http”) and an “authority”, typically an IP address or hostname, and perhaps a port number (e.g., 10.2.12.45:8080). The algorithm used by an ATSC 3.0 receiver to generate the portion of a URI that determines the origin of a broadcast file shall conform to the restrictions specified below. Note that a resource from a broadband source has an origin defined by the web server hosting the resource.

The URI provided by the Receiver shall have the following form for any resource within an Application Context Cache:

<baseURI>/<pathToResource>

where:

<baseURI> is defined as <scheme>://<authority>[/<pathPrefix>]

<scheme> and <authority> are standard URI elements as defined in RFC 3986 [21],

<pathPrefix> is an optional path prefix prepended to the path; and

<pathToResource> is the relative path supplied by ROUTE (e.g., Content-Location from the EFDI [1])

As described in Section 6, when delivered via broadcast, a Broadcaster Application is launched with a URI defined by the Receiver. The Base URI portion of the URI, that is, the portion of the URI prior to the relative path supplied by ROUTE transmission, shall uniquely correspond to the Application Context Identifier.

The algorithm used to generate the Base URI for a given Application Context Identifier should be consistent, that is, the algorithm should repeatedly produce the same URI over time. Thus, the combination of scheme, authority and path prefix should always be unique. However, Broadcaster Applications should not rely on the uniqueness by itself of either the origin or path prefix provided by the Receiver as part of the Entry Page URL. The Broadcaster Application is expected to manage a local name space for setting cookies and other local User Agent storage elements.

If the current Application Context Identifier remains the same, even though the Entry Page may change, all of the associated resources shall continue to be available within the Application Context Cache through the Receiver Web Server. Entry Page changes are signaled by application signaling as described in Section 6.3.1.

If the Application Context Identifier changes, the Receiver may reuse the Application Context Cache previously created for that Application Context Identifier or a new cache may be created. The Receiver may elect to maintain any previous Application Context Cache, albeit unknown to Broadcaster Applications with differing Application Context Identifiers, on the presumption that these previous Application Context Caches may be needed soon. Alternatively, the Receiver can free the resources associated with the previous Application Context Cache. If a file is not cached, the Receiver Web Server may respond to the request by waiting for the next delivery of the file or with an error code. File caching decisions are left entirely to the Receiver implementation; however, attributes associated with Application Context Cache files are intended to provide prioritization information to the Receiver caching mechanisms (see Section 6.2).

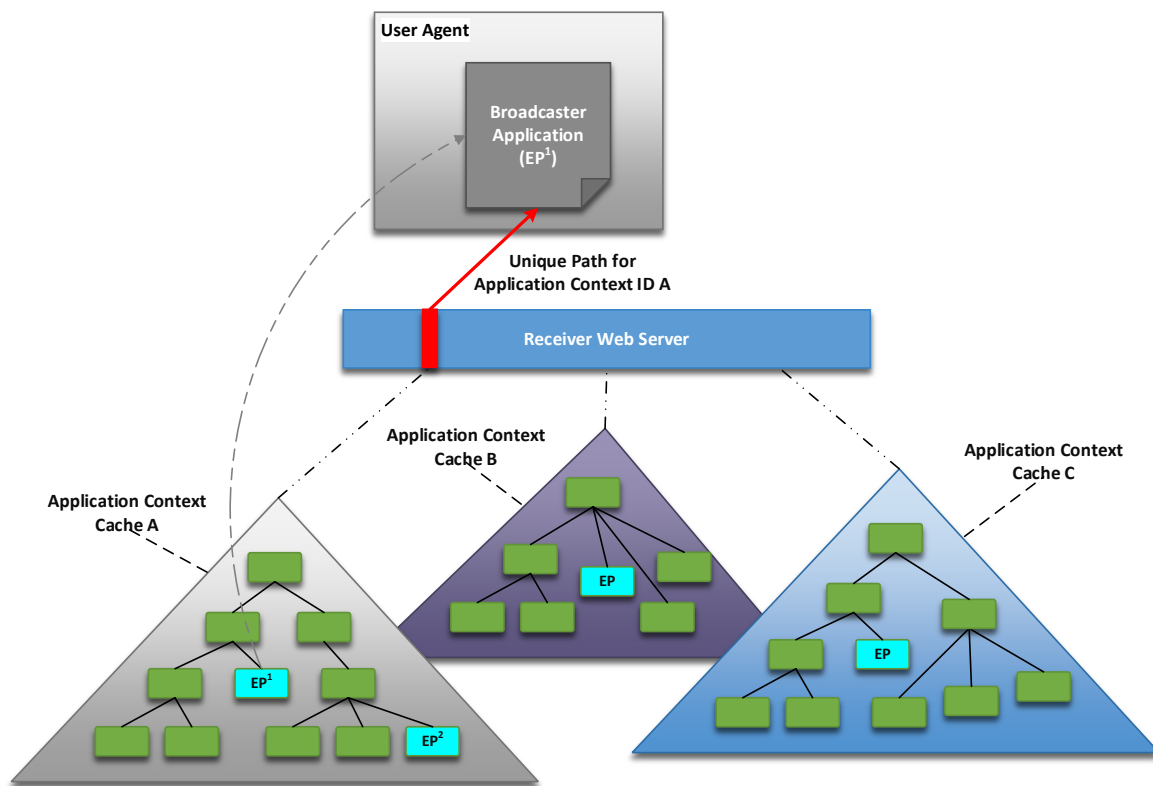


Figure 5.2 Application Context Identifier Conceptual Model.

Figure 5.2 provides a conceptual model of how Application Context Identifiers are related to the Broadcaster Application and broadcast files. The diagram provides an example of how resources (files and directories) are made available to a Broadcaster Application through the Receiver Web Server using URIs unique to a given Application Context Identifier. In the figure, the Broadcaster Application is shown operating in the User Agent having been launched using Entry Page, **EP¹**. At some point while **EP¹** is active, application signaling could launch the Entry Page designated as **EP²**. In this case, the Application Context Cache A and access to it would remain constant with the User Agent loaded with **EP²**. The Receiver may or may not provide access to the other Application Context Caches corresponding to different Application Context Identifiers. Broadcaster Applications should restrict access to resources within their own Application Context Cache as provided by the Receiver, or to the Internet if broadband is available.

Broadcaster Applications delivered on services spanning multiple broadcasts may have the same Application Context Identifier allowing Receivers with extended caching capabilities to maintain resources across tuning events. This allows broad flexibility in delivering resources on multiple broadcasts for related Broadcaster Applications.

6. BROADCASTER APPLICATION MANAGEMENT

6.1 Introduction

A Broadcaster Application is a set of documents comprised of HTML5, JavaScript, CSS, XML, image and multimedia files that may be delivered separately or together within one or more packages.

This section describes how a Broadcaster Application package is:

- Downloaded,
- Signaled,
- Launched, and
- Managed.

Additionally, it describes how a Broadcaster Application can access the resources made available by the Receiver.

Figure 6.1 diagrams the relationships between various concepts within a generalized reference receiver architecture—whether distributed, i.e., the Receiver Web Server is in a separate physical device from the User Agent, or not. It is not intended to define a particular receiver implementation but to show relationships between the various elements discussed in this section.

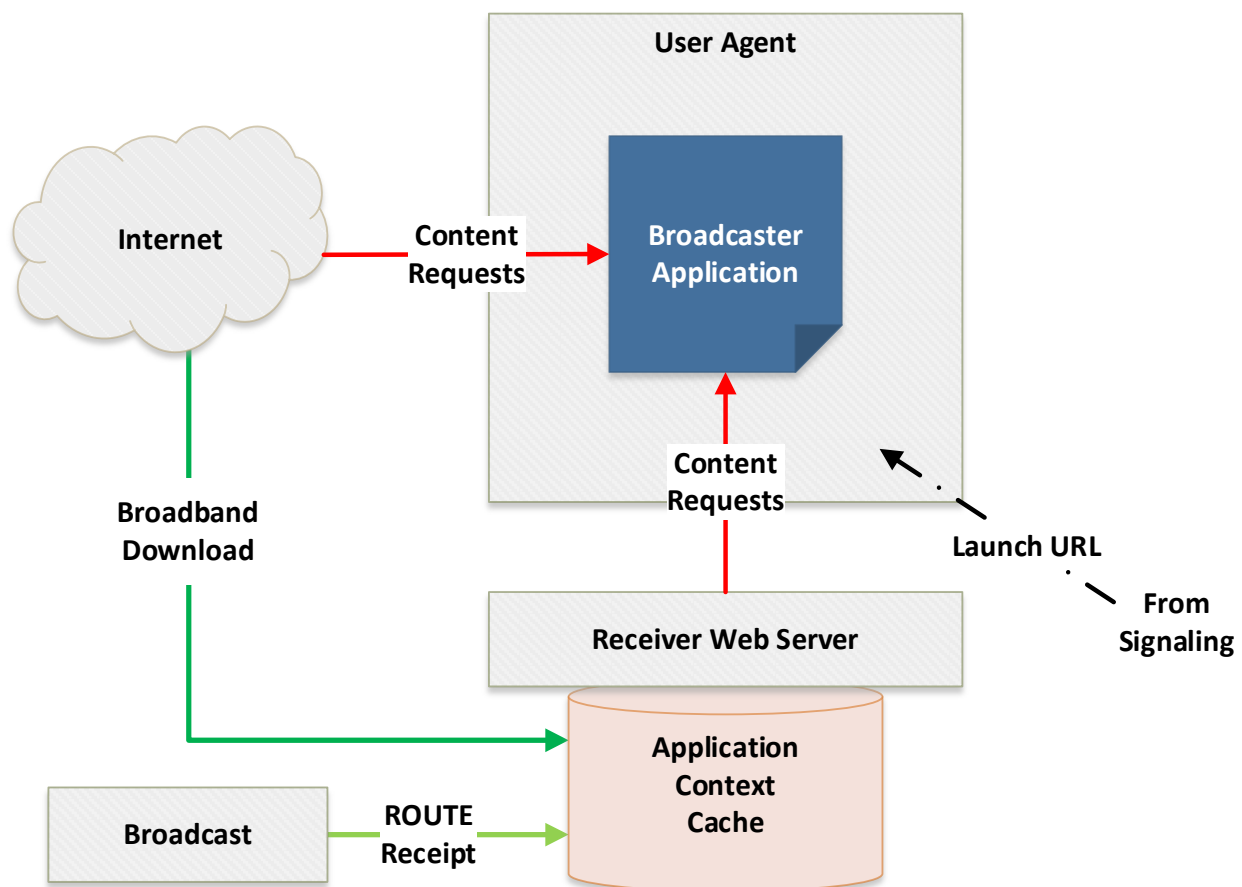


Figure 6.1 Receiver Conceptual Architecture.

The Broadcaster Application is launched after the Receiver receives application signaling information (see Section 6.3 below) and then forwards the launch URL to the User Agent, which, in turn, loads the Broadcaster Application Entry Page from the URL. Note that the URL may point to an Internet server or to the Receiver Web Server depending on how it is formatted in the service application signaling, specifically, the `HTMLEntryPackage@bcastEntryPageUrl` or `HTMLEntryPackage@bbandEntryPageUrl` attributes of the HELD [4]. The specific mechanism of communicating the Broadcaster Application entry URL to the User Agent is a receiver implementation detail. However, the entry URL has specific arguments that must be provided as described in Section 8.2.

Once the main Broadcaster Application Entry Page has been loaded, it may begin requesting content from various local or external URLs. This may be done through JavaScript or standard HTML5 href requests in the W3C-compliant fashion. It is assumed that any content received over broadcast via ROUTE file delivery is available through the Application Context Cache and accessed using the Receiver Web Server. This specification makes no assertions as to how this is done nor how any cache or storage is implemented. It does, however, describe how the Broadcaster Application can access the resources using HTTP requests to the Receiver Web Server.

Note that the User Agent supports various local W3C storage mechanisms according to Section 5.2. The User Agent may also perform internal caching of content. The internal W3C-compatible storage mechanisms implemented within the User Agent should not be confused with the Application Context Cache shown separately in Figure 6.1. The Broadcaster Application can use standard W3C interfaces to discover and use the various User Agent storage facilities.

6.2 Application Context Cache Management

6.2.1 Signaling Intent for File Caching

All files delivered over broadcast to the Application Context Cache are carried in multipart/signed packages as described in A/331 [1] and required by Section 6.5 of this standard. From the ROUTE standpoint, each package is an opaque file object so the subsequent File elements within the FDT-Instance element of the EFDT describe only the multipart/signed package object.

A main header is defined within the multipart/signed package that describes various parameters including the necessary boundary text that delineates files within the package. The file data resides within these boundary-separated blocks which, in turn, include header elements, referred to herein as a boundary header, prior to the individual file data, that can provide metadata specific to the file within the package block.

To provide a manifest for files contained within the package, a `metadataEnvelope`, as defined in A/331 Section 7.1.6, shall be included as the first object in the package. Content shall not be embedded in the `metadataEnvelope`; the “referenced” mode shall be used.

Within the `metadataEnvelope` fragment, a `metadataEnvelope.item` element shall be present corresponding to each file within the package. The `metadataEnvelope.item` attributes shall be interpreted as follows:

- The required `@metadataURI` attribute shall provide the relative path of the file referenced by this `metadataEnvelope.item`. The URI value shall match the relative path supplied in the `Content-Location` parameter included as part of the boundary header for the referenced file. This will provide the relative path of the file within the Application Context Cache.

- The `@version` attribute increments when a new version of the referenced file has been provided in the package. The Receiver shall rely on the `@validFrom` attribute when detecting file version changes and can safely ignore the `@version` attribute.
- The `@validFrom` attribute shall be required and shall indicate when the referenced file was last modified. A new version of a file shall be signaled by updating this time stamp within the `metadataEnvelope.item` associated with the file. This value shall be provided as the `Last-Modified` HTTP header parameter supplied when accessing the file through the Receiver Web Server unless the boundary header contains an `ATSC-HTTP-Attributes` parameter as described in Section 6.2.1.1 that overrides this default. This value shall be used when calculating the age of the file.
- The date and time supplied in the optional `@validUntil` attribute shall be used to indicate when the file is no longer needed and can be released from the Application Context Cache. The Broadcaster Application can read the expiration time of the file using the `Expires` HTTP parameter supplied when accessing the file through the Receiver Web Server unless the boundary header contains an `ATSC-HTTP-Attributes` parameter as described in Section 6.2.1.1 that overrides this default.
- The required `@contentType` attribute shall provide the MIME type of the referenced file. This attribute shall match the `Content-Type` value defined as part of the boundary header within the package, if provided. Note that the `EFD.T.FDT-Instance.File` also contains a `Content-Type` definition, but this should always be `multipart/signed` indicative of the referenced package object. This value may be accessed through the `Content-Type` HTTP header parameter supplied when accessing the file through the Receiver Web Server.

This specification defines an additional attribute that extends the `metadataEnvelope.item` specification defined in A/331 [1]. Table 6.1 provides an informative definition of the ATSC extended attribute when included within a signed package destined for the Application Context Cache. The normative semantics of the attribute are provided below the table.

Table 6.1 ATSC-Defined Extension to the `metadataEnvelope.item` Element

Attribute Name	Cardinality	Data Type	Description
<code>@contentLength</code>	0..1	long	Provides the length in bytes of the referenced file. This value may be accessed through the <code>Content-Length</code> HTTP attribute in a response to a User Agent request.

`@contentLength` – The optional `@contentLength` attribute shall define the length in bytes of the referenced file within the package. When defined, the length value shall be made available as part of the `Content-Length` HTTP header element provided by the Receiver Web Server when the referenced file is accessed.

6.2.1.1 Boundary Header HTTP Attribute Definition

A boundary header element, `ATSC-HTTP-Attributes`, may optionally be supplied in the boundary header of a file within the `multipart/signed` package. This element provides a list of HTTP header elements that shall be provided whenever the associated file is requested through the Receiver Web Server. The syntax of this attribute shall be defined as follows:

```
attributes := "ATSC-HTTP-Attributes" ":" parameter[";" parameter]*
```

where each `parameter` is an HTTP header field as described by RFC 7230 [13] except that the colon, “:”, used by the HTTP header field shall be replaced with an equals sign, “=”, to comply with constraints imposed by the multipart standard, RFC 2045 [16].

6.2.2 Application Context Cache Hierarchy Definition

All interactive content is carried in signed packages of files and transmitted via ROUTE (Section 6.4). All signed packages whose application signaling denotes a particular Application Context Identifier shall be provided in a single hierarchy accessible to the Broadcast Application with a unique Base URI. The Base URI concept and its semantics are described in Section 5.3.

The choice of whether a package or file is immediately stored to the Application Context Cache on receipt within the ROUTE file stream or if the Receiver chooses to defer storage until the particular broadcast data element is referenced is a Receiver implementation decision. However, if the underlying Receiver Web Server cannot provide the requested content, an HTTP status code within the 400-series `Client Error` or 500-series `Server Error` is returned indicating that some error condition occurred [15]. Typically, this is either 404 `Not Found`, 408 `Request Timeout` or 504 `Gateway Timeout error`; however, Broadcaster Applications should be constructed to deal with any HTTP status code when referencing resources not contained within their Entry Package.

Similarly, the present document does not specify how frequently packages needed for the Broadcaster Application are transmitted nor how frequently application signaling metadata is sent. These decisions depend on several factors such as what time the Broadcaster Application functionality is actually needed during the program, how quickly the Receiver needs to access the Broadcaster Application after the service is selected, and the overall bandwidth needed to carry Broadcaster Application resources, to list a few. These and other factors are likely different for every Broadcaster Application depending on its overall purpose. The Application Signaling standard, A/331 [1], requires broadcast packages to be available when the HELD signals the Broadcaster Application. In addition, A/331 also defines a Distribution Window Description fragment (DWD) that provides a mechanism to signal when packages will be available in the broadcast at times other than when the Broadcast Application is signaled in the HELD.

The broadcaster shall be responsible for defining and managing any hierarchy below the Application Context Cache root directory through use of the directory and path mechanisms described in Section 6.2.1 above. Any hierarchy below the Application Context Cache Base URI level is up to the broadcaster to define.

An example of how such a hierarchy could be defined is described below. Figure 6.2 shows an example of such a hierarchy for **BaseURI B**.

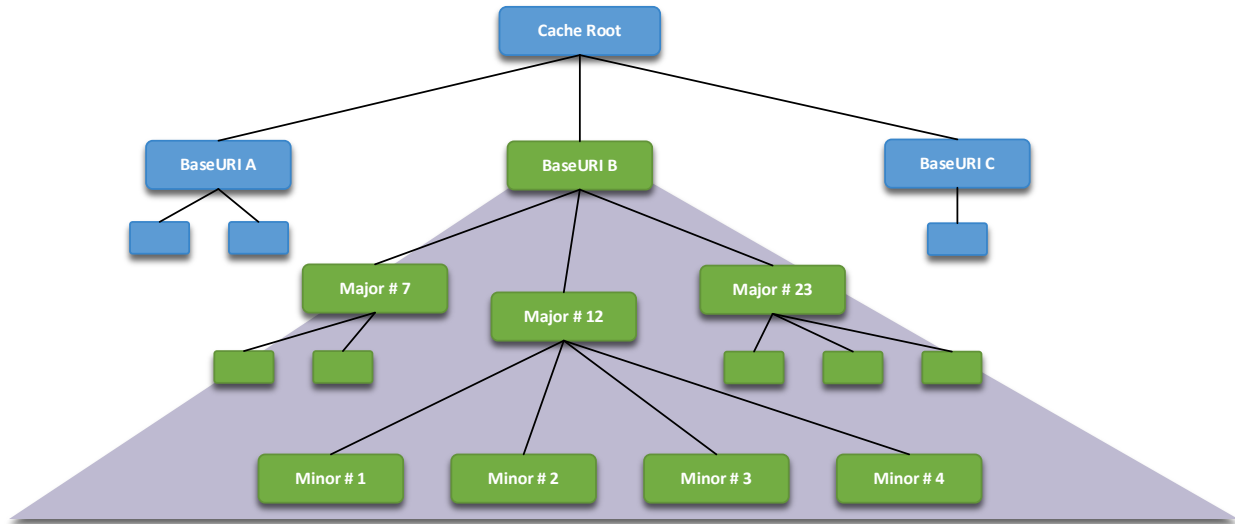


Figure 6.2 Example Application Context Cache Hierarchy.

As an example, presume that the broadcaster transmitted the ROUTE data such that the Broadcaster Application files are placed within the “Minor #2” directory in the hierarchy shown in Figure 6.2. Further, assume that the Broadcaster Application Entry Page is called “main.html”. To launch the Broadcaster Application, the application signaling in this example provides the relative URI of “12/2/main.html” in the HTMLEntryPackage@bcastEntryPageUrl attribute of the HELD [4]. Note that the local path is a convention defined by the broadcaster—it could just as easily have been called “red/green”. The actual URL of the Broadcast Application would be:

<BaseURI>/12/2/main.html

Note that the Receiver-created **<BaseURI>** portion of the URL is highlighted as bold. The Broadcaster Application would be able to reference any directory corresponding to its AppContextID hierarchy as designated by the green boxes in Figure 6.2 under the “BaseURI B” box.

The Receiver provides the Base URI information associated with the current Application Context Identifier to the Broadcaster Application through a Receiver WebSocket Server API (see Section 9.2.7). Note that this Base URI is available through the normal W3C document object model (DOM) if the Broadcaster Application is sourced from broadcast. In this case, the Broadcaster Application can simply access content using relative URLs or constructing full URLs using `Document.location`. The WebSocket API is most applicable for Broadcaster Applications hosted from broadband allowing them to gain access to resources received over broadcast.

6.2.3 Active Service Application Context Cache Priority

Once the packaged resources, known as the Entry Package, of the Broadcaster Application (see Section 6.3) have been acquired and placed in the Application Context Cache, the Broadcaster Application can assume that the Entry Package resources, that is, those files delivered along with the Broadcaster Application Entry Page in a single package, shall remain available as long as the Broadcaster Application is loaded in the User Agent. If the Receiver cannot provide the entire Entry Package to the Broadcaster Application reliably, the Broadcaster Application shall not be launched. This allows broadcasters to determine the required files of their Broadcaster Application and send them all as one package thereby guaranteeing that those files will be available if the

Broadcaster Application has been launched. In addition, Receivers are expected to make every effort to cache files delivered with a particular service for the Broadcaster Application(s) associated with the current service Entry Package.

Note that the Receiver may have to release other portions of the cache that are not active to accommodate files in the currently active hierarchy. Indeed, it may be necessary for the Receiver to purge the entire cache to accommodate the present active service. Since the user actively selected the current service, the Receiver assumes that the current service content preempts all other content from the user perspective.

A Broadcaster Application may limit the amount of cache required by using the Filter Codes mechanism. Filter Codes provide a way to differentiate application content allowing a Broadcaster Application to indicate which packages it wishes to be placed in the Application Context Cache and which packages it wishes to ignore. A set of WebSocket APIs is provided to allow the Broadcaster Application to manage the Filter Codes (see Section 9.11).

A Broadcaster Application can mark content as unused to hint to a receiver that the resources are no longer needed. A WebSocket API is provided (see Section 9.9) that allows files or entire directory hierarchies to be marked as unused. Subsequent access to resources that are marked as unused shall result in an error. The HELD (A/331 [1]) also provides an attribute, `@clearAppContextCacheDate`, associated with the Broadcaster Application that indicates that any file in the Application Context Cache created before the specified date and time should be removed. This enables older files present in an Application Context Cache to be marked as obsolete.

6.2.4 Cache Expiration Time

The `@validuntil` attribute of a file, as defined in Section 6.2.1, shall indicate that the broadcaster expects the file to remain in the Application Context Cache until the expiration time has been reached regardless of whether the Broadcaster Application is currently loaded in the User Agent. However, the storage requirements of the loaded Broadcaster Application take precedence over the `@validuntil` attribute, so the Receiver may be forced to release files from other services prior to their `@validuntil` time to provide storage to the current service. The Broadcaster Application must be prepared to deal with the possibility that a resource may not be available. Receivers may elect to leave the resource within the Application Context Cache or purge it whenever is convenient after the `@validuntil` date and time has been passed, depending on other cache management criteria.

Note that for resources that are part of the Entry Package, the Broadcaster Application can assume that those files are available at startup and remain available if the Broadcaster Application is running regardless of the `@validuntil` time as described in Section 6.2.3.

6.2.5 Advanced Emergency Alert Enhancement Content Considerations

The AEAT may reference AEA enhancement content through URLs. This content is delivered either in the broadcast as a separate ROUTE stream, in which case the referencing URL will be relative, or over broadband where a fully qualified URL will be provided.

In the broadcast case, the broadcaster is responsible for providing the Application Context Id of any Broadcaster Applications that need to access the AEA enhancement content as part of the EFDT fragment describing the signed package containing the AEA enhancement content. The Receiver shall treat this content as it would normal ROUTE-delivered content by making it available in the Application Context Caches corresponding to the listed Application Context IDs. Note that since the AEA enhancement content will occupy the same hierarchy as other ROUTE-

delivered data, the broadcaster must ensure that no conflicts occur when defining the corresponding file hierarchies.

Similarly, any cache management performed by the Broadcaster Application should take the life span of the AEA event into account when signaling `@clearAppContextCacheDate` or using the Mark Unused API (see Section 9.9) to indicate the content is no longer needed. Any broadcast NRT data associated with an AEA event is expected to have a `@validuntil` attribute defined to be at least the duration of the AEA event.

6.3 Broadcaster Application Signaling

The ATSC 3.0 Receiver is responsible for retrieving the Broadcaster Application files and resources from the location and transport mechanism indicated by ATSC 3.0 application signaling [4]. A service must be successfully selected and, for broadcast resources, where a relative URI is supplied, the Entry Package shall be completely available within the Application Context Cache (Section 6.2.2) for the ATSC 3.0 Receiver to launch the Broadcaster Application. For external broadband references where a full URL is provided, the URL supplied in the `HTMLEntryPackage@bbandEntryurl` attribute for the Broadcaster Application section of the HELD [1] shall be launched directly.

6.3.1 Broadcaster Application Launch

When a new service is selected and a Broadcaster Application entry URL is present in the application signaling for that service, there are two possible conditions to consider:

- 1) **No Current Broadcaster Application** – When there is no Broadcaster Application currently active in the User Agent, the Receiver shall launch the Broadcaster Application specified by the relative URI in the application signaling for the new service once the complete Entry Package has been received and the Application Context Cache is available for access.
- 2) **Current Broadcaster Application** – When there is a Broadcaster Application previously loaded, the URI and the `AppContextID` from the present service signaling matches, then the same Broadcaster Application has been requested. The current Broadcaster Application shall receive a notification that a new service has been selected via the API described in Section 9.3.3. It is up to the Broadcaster Application design whether to reload its Entry Page and restart or remain on the currently active page.

If the Broadcaster Application URI or the `AppContextID` from the newly selected service application signaling does not match the presently loaded Broadcaster Application URI or the `AppContextID`, the new Broadcaster Application shall be launched as if no previous Broadcaster Application was loaded.

6.3.2 Broadcaster Application Events (Static / Dynamic)

Actions to be taken by Broadcaster Applications can be initiated by notifications delivered via broadcast or broadband or, in a redistribution setting, via watermarks. A/337 [4] uses the term “Events” for such notifications.

Broadcast delivery of events is defined in Section 4.1 of A/337 [4] including delivery for ROUTE/DASH-based services and MMT-based services.

Broadband delivery of events is defined in Sections 4.2 and 4.5 of A/337 [4] including delivery for ROUTE/DASH-based services and MMT-based services.

Both the broadcast and broadband delivery of events defined in A/337 [4] support batch (static) and incremental (dynamic) delivery.

In a redistribution setting, events can be also delivered via video and audio watermarks as described in Section 4.3 of A/337 [4].

Detailed specification of the WebSocket APIs used to register for and receive event stream notifications is provided in Section 9.6.

6.4 Broadcaster Application Delivery

The file delivery mechanism of ROUTE, described in A/331 [1], provides a means for delivering a collection of files either separately or as a package over the ATSC 3.0 broadcast. The ROUTE-delivered files are made available to the User Agent via a Receiver Web Server as described in Section 6.2. The same collection of files can be made available for broadband delivery by publishing to a receiver-accessible web server. Furthermore, the application signaling [4] determines the source of the Broadcast Application Entry Page, and the location of any other files and packages that are delivered by broadcast.

- 1) A relative Entry Page URI indicates that the source of the Broadcaster Application Entry Page is broadcast ROUTE data.
- 2) An absolute Entry Page URL indicates that the Broadcaster Application Entry Page should be sourced from broadband.
- 3) If any files or packages are delivered by broadcast, the application signaling identifies the LCT channels that are used to deliver the files and/or packages.

Note that while the initial Entry Page may be sourced from either broadband or broadcast according to the `HTMLEntryPackage@bbandEntryPageUrl` or `HTMLEntryPackage@bcastEntryPageUrl` attributes in the HELD, there is no constraint regarding using either broadcast or broadband resources within the Broadcaster Application itself. Hybrid delivery of Broadcaster Application files is allowed and expected.

6.4.1 Broadcaster Application Packages

The file components comprising the Broadcaster Application shall be delivered within one or more multi-part MIME packages using ROUTE or over broadband as individual files using HTTPS. All files made available through the Receiver Web Server shall be delivered to the receiver as signed packages as described in A/331 [1].

It is not required that all resources used by a Broadcaster Application be delivered in a single ROUTE package when delivered over broadcast. The broadcaster may choose to send a relatively small Entry Page in a signed Entry Package which then performs a bootstrapping operation to determine what other resources have been delivered or are accessible via the broadcast delivery path and, in turn, which resources need to be obtained using broadband requests. Since the Entry Package containing the Entry Page shall be received in its entirety before launching the Broadcaster Application (per Section 6.2.3), the Broadcaster Application can forgo any checks for basic resources and perhaps speed the initial startup time. It is conceivable that Broadcaster Applications may have incremental features based on the availability of resources on the Receiver. In other words, the Broadcaster Application may add features and functions as more resources are available. Control for selecting specific signed packages to be made available is provided using the Filter Codes API (see Section 9.11).

In addition, the Broadcaster Application may request resources and content from or perform other activities with broadband web servers making the Broadcaster Application a true Web Application in the traditional sense. A Broadcaster Application should be aware that all Receivers

may not contain sufficient storage for all the necessary resources and may not have a broadband connection.

6.4.2 Broadcaster Application Package Changes

Broadcaster Application resource files and packages may be updated at any time. Mechanisms for determining that a new file or package is being delivered are defined in FLUTE, which is the underlying standard used by ROUTE [1]. The broadcaster may send an Event Stream notification to let the Broadcaster Application know that something has been changed. The Broadcaster Application determines how such changes should be addressed based on the Event Stream notification.

6.5 Security Considerations

All Broadcaster Application files delivered over the broadcast shall be delivered using the ROUTE Signed Package mechanism described in A/331 Section A.3.3.5 [1] and A/360 Section 5.2 [6]. A/331 describes the encapsulation of Broadcaster Application files in MIME multipart packages while A/360 describes the encapsulation of that MIME multipart package into an S/MIME wrapper to secure the Broadcaster Application files. Files are deemed to be part of the Broadcaster Application if they will be accessible from the Application Context Cache through the Receiver Web Server interface. For Receivers that support signing, it is expected that they will only make content from correctly signed packages available through the Receiver Web Server interface.

The Broadcaster Application files may be delivered over broadcast in as many signed packages as desired – there is no restriction on dividing files among signed packages. In fact, it may be typical that core functions of the Broadcaster Application are delivered in the Entry Package while extended content and functionality are delivered in separate packages. In addition, the Filter Codes mechanisms (see Section 9.11) can be used to select various packages as user preferences or other selection criteria are discovered. Regardless of how files are partitioned into separate packages, these packages must be signed per the requirement in the previous paragraph.

Broadcaster Application files delivered over broadband shall be secured using standard W3C mechanisms. All connections to broadband servers shall use a secure connection as described in A/360 Section 5.1 [6]. The content received over broadband using a secure connection shall be considered trusted as if it had been received in a signed package over broadcast.

6.6 Companion Device Interactions

The ATSC 3.0 Companion Device standard (A/338 [29]) specifies how a separate device, known as the Companion Device (CD), interacts with the Receiver, known as the Primary Device (PD) in the A/338 standard. The A/338 standard extends the APIs defined in Sections 8 and 9 of the present standard to provide CD Manager APIs that allow the Broadcaster Application to discover and launch CD Applications operating on the Companion Device. These CD Manager APIs also provide a mechanism for the Broadcaster Application to obtain WebSocket service end points that allow application-to-application communication between the CD Applications and the Broadcaster Application. The Broadcaster Application may support multiple connections by requesting multiple end points. To use the CD Manager APIs, the Broadcaster Application can obtain the WebSocket URL with the mechanism defined in Section 8.2.1.

Complete details of the Companion Device Interactions regarding the discovery and launch of Companion Device applications, application-to-application WebSocket communication mechanisms, and Receiver WebSocket APIs available to the CD Application can be found in the ATSC 3.0 Companion Device standard (A/338 [29]).

7. MEDIA PLAYER

In the ATSC 3.0 Receiver environment, there are two software components that can play out media content delivered via either broadcast, broadband, or redistribution. For the purposes of this specification, these two logical components are referred to as Application Media Player (AMP) and Receiver Media Player (RMP), and these are described further in this section. The AMP is JavaScript code (e.g., DASH.js), which is part of an HTML5 Broadcaster Application, while the RMP is receiver-specific implementation. The AMP uses the video tag and MSE to play out media content regardless of the content origination or delivery path. Details of the RMP design and implementation are out of scope for this specification and any design descriptions provided in this specification are only as informative reference. Whether AMP or RMP is used to play out a media content, there are several use cases:

- **Broadcast or Hybrid Broadband / Broadcast Live Streaming** – The content segments arrive either via broadband or broadcast.
- **Broadband Media Streaming** – Media content streaming over broadband (on-demand or linear service).
- **Downloaded Media Content** – Media content downloaded over broadcast or broadband ahead of time. Details of how media content is downloaded over broadband or broadcast is described in Section 9.3.10 of this specification.

The type of media streams played depends on signaling in the MPD of live broadcast streams, or specific Broadcaster Application logic.

The DASH Client specification [32] provides the expectations for behavior of such players and is not further described here.

In redistribution scenarios, media content arrives using a method that does not employ ATSC 3.0 protocols (e.g. HDMI). Redistribution services are presented by an RMP.

7.1 Utilizing RMP

The RMP can be triggered to play out media content streamed over broadcast by receiver logic or by an explicit request from a Broadcaster Application. These distinctions are further described in this section.

7.1.1 Broadcast or Hybrid Broadband and Broadcast Live Streaming

When tuned to a new service, the RMP determines whether to play out the media stream or whether to wait for the Broadcaster Application to determine whether to play out the media stream. The HTML Entry page Location Description (HELD) specified in A/331 [1] signals which entity (AMP, RMP) is intended to play the media stream; however, receiver logic can choose to play out the media stream, regardless of what is signaled in HELD. The information in the MPD determines whether the media stream segments are to be played out from broadcast or from a combination of broadband and broadcast.

7.1.2 Broadband Media Streaming

The RMP can play out a service delivered by broadband media streaming if the service signaling indicates a broadband MPD URL, or if the Broadcaster Application requests that the RMP play out the stream. For the purposes of this document, there is no distinction between play-out of live broadband streaming vs. on-demand over broadband. The differentiation on how the MPD is organized for these two use cases is described further in the DASH Client specification.

7.1.3 Downloaded Media Content

Depending on the request from the Broadcaster Application, the RMP can play out downloaded media content that was delivered via broadband or broadcast. The Broadcaster Application can make such a request using the Set RMP URL WebSocket API as described in Section 9.7.3.

7.1.4 Redistribution

Interactive content can be supported in redistribution scenarios. The Receiver can obtain the ATSC 3.0 service from a redistribution source, present the service using an RMP, acquire service and application signaling as described in A/336 [3], and acquire the application package via broadband as described in Section 6.4.

7.2 Utilizing AMP

7.2.1 Broadcast or Hybrid Broadband and Broadcast Live Streaming

Although broadcast or hybrid live media streaming is typically played out by the RMP, it is possible for the AMP to request playback of the content. A flag in the HELD indicates whether the RMP can immediately play out the media content, or whether the service expects the AMP to play out the live media streaming. The Receiver can ignore this signaled expectation, in which case the RMP can immediately play out the live media streaming. There are two possible methods on how an AMP can play out a live media streaming, Pull and Push, and they are described in Sections 7.2.4 and 7.2.5.

7.2.2 Broadband Media Streaming

There is no special consideration for playing media streams delivered over broadband-only other than what is provided in the DASH client specification.

7.2.3 Downloaded Media Content

The AMP can play out broadband or broadcast downloaded media content. The Broadcaster Application discovers the MPD URL of the downloaded media content, and initiates play-out using one of the two mechanisms described here and described in Section 9.7.3.

7.2.4 AMP Utilizing the Pull Model

The Pull model behaves the same as if the Receiver was a remote DASH Server. The DASH specification describes the details on how a DASH server should be implemented.

7.2.5 AMP Utilizing the Push Model

In the Push model, the AMP opens the binary WebSocket connections specified in Section 8.2.1. Opening these WebSocket connections is an implicit request that the Receiver retrieve the Initialization and Media Segments of the media content and pass them to the AMP via these connections. The AMP then uses an HTML5 <video> tag in conjunction with MSE to pass the media content to the Receiver's decoders for decoding and presentation. In the broadcast case, the media are retrieved from the broadcast via the ROUTE Client and pushed to the AMP via the WebSocket server. In the broadband case, the media are retrieved from a remote HTTP server via the HTTP Client and pushed to the AMP via the WebSocket server. In both cases, a DASH Streaming Server acts as an intermediary to retrieve the media segments from the ROUTE Client or HTTP Client and stream them to the Broadcaster App via the WebSocket connections.

8. ATSC 3.0 WEB SOCKET INTERFACE

8.1 Introduction

A Broadcaster Application on the Receiver may wish to exchange information with the Receiver platform to:

- Retrieve user settings
- Receive an event from the Receiver to the Broadcaster Application
 - Notification of change in a user setting
 - DASH-style Event Stream event (from broadcaster)
- Request receiver actions

In order to support these functions, the Receiver includes a web server and exposes a set of WebSocket Remote Procedure Call (RPC) calls. These RPC calls can be used to exchange information between a Broadcaster Application running on the Receiver and the Receiver platform. Figure 8.1 shows the interaction between these components.

In the case of a centralized receiver architecture, the Receiver Web Server typically can be accessed only from within the Receiver by Broadcaster Applications in the User Agent.

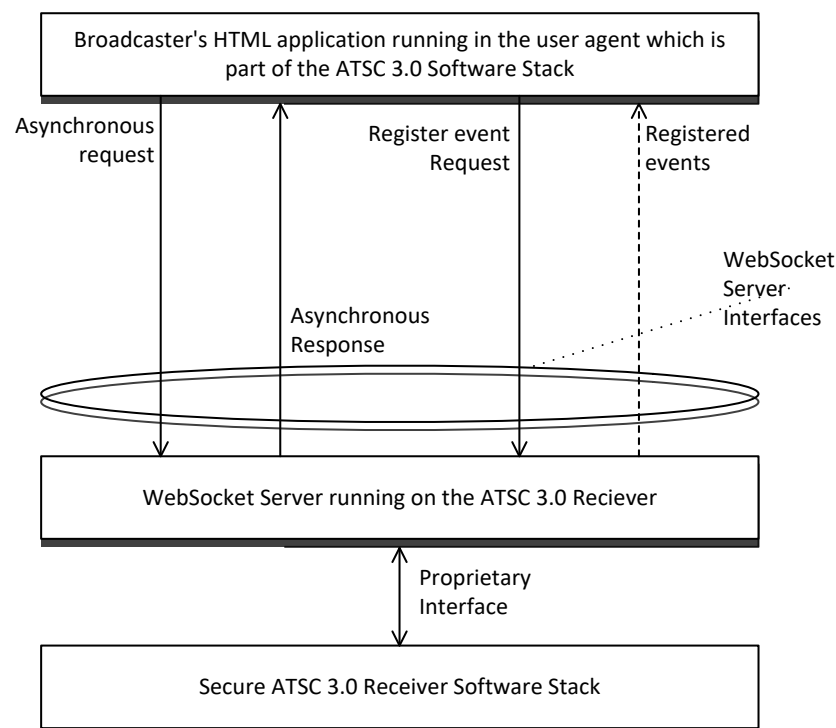


Figure 8.1 Communication with ATSC 3.0 receiver.

One or more ATSC 3.0 WebSocket interfaces are exposed by the Receiver. All Receivers support a WebSocket interface used for command and control. Some Receivers also support three additional WebSocket interfaces, one each for video, audio, and caption binary data. The Broadcaster Application or companion devices can connect to the command and control interface to retrieve state and settings from the Receiver and perform actions, such as change channels.

8.2 Interface Binding

Since the APIs described here utilize a WebSocket interface, the Broadcaster Application can rely on standard browser functionality to open the connection and no specific functionality needs to be present in the Broadcaster Application.

In order to communicate with the WebSocket server provided by the Receiver, the Broadcaster Application needs to know the URL of the WebSocket server. The WebSocket server location may be different depending on the network topology (e.g., integrated vs. distributed architecture), or it may be different depending on the Receiver implementation. In order to hide these differences from the Broadcaster Application, the Broadcaster Application Entry Page URL is launched with a query term parameter providing information regarding the location of the Receiver WebSocket Server.

When an Entry Page of a Broadcast Application is loaded on the User Agent, the URL shall include a query term providing the Base URI of the ATSC 3.0 WebSocket Interface supported by receivers. Similarly, the Receiver shall report the current version of the WebSocket API supported by providing another query term containing the release date of this standard. Using the ABNF syntax, the query component shall be as defined below:

```
query = (wsQuery "&" revQuery) / (revQuery "&" wsQuery)
```

```
wsQuery = "wsURL=" ws-url
```

```
revQuery = "rev=" yyyyymmdd
```

The `ws-url` is the base WebSocket URI and shall be as defined in RFC 6455 [20]. The `yyyyymmdd` value shall contain the year (`yyyy`), month (`mm`) and day (`dd`) when the present standard was released. For example, the first release of this standard was 18 December 2017. That value is represented as '20171218'. The date used for any given release shall be taken from the corresponding entry in the 'Date' column of the Revision History table at the beginning of this document.

The following shows an example of how such a query string is used to launch the Broadcaster Application. In this example, if the Entry Page URL is:

```
http://localhost/xbc.org/x.y.z/home.html
```

and the WebSocket APIs are based on the revision of the standard as released on 20 July 2018, the Broadcaster Application is launched as follows:

```
http://localhost/xbc.org/x.y.z/home.html?wsURL=wss://localhost2:8000
&rev=20180720
```

The `wsURL` and `rev` query parameters are added to load an entry page URL of a broadcast-delivered application. It is expected that a broadband web server would ignore a `wsURL` query parameter in the URL of an HTTP request if it were to appear. The `rev` query term is applicable to launching Broadcaster Applications from both broadcast and broadband.

8.2.1 WebSocket Servers

All Receivers shall support access to a WebSocket interface used for communication of the APIs described in Section 9. Receivers which support push-mode delivery of binary media data (video,

audio, and captions) also support three additional WebSocket interfaces, one for each type of media data. Receivers that support the A/338 Companion Device standard also provide an additional WebSocket interface that allows communication with the CD Manager within the Receiver (see Section 6.6). Table 8.1 describes the four interfaces. In the table, the term “*WSPath*” represents the value of the *wsURL* parameter discovered in the procedure above.

Table 8.1 WebSocket Server Functions and URLs

WebSocket Interface Function	URL	Receiver Support
Command and Control	<i>WSPath/atscCmd</i>	Required
Video	<i>WSPath/atscVid</i>	Optional
Audio	<i>WSPath/atscAud</i>	Optional
Captions	<i>WSPath/atscCap</i>	Optional
Companion Device	<i>WSPath/atscCD</i>	Optional

If an optional WebSocket URL shown in Table 8.1 is not supported, the Receiver shall respond with the HTTP status code “404 Not Found” when the Broadcaster Application attempts to connect to the optional interface. Receipt of this status results in the failure of the WebSocket connection to the particular WebSocket API.

In the push model, each MPEG DASH Media Segment file delivered via a Video/Audio/Captions WebSocket interface is delivered in a binary frame of the WebSocket protocol. The command and control interface uses text frame delivery.

8.2.1.1 Initializing Pushed Media WebSocket Connections

Upon establishment of any of the media WebSocket connections listed in Table 8.1 (*atscVid*, *atscAud*, *atscCap*), it is expected that the first data sent by the broadcast receiver over such a connection is a text message (opcode 0x1, as defined in Section 5.2 of IETF RFC 6455 [20]) with the payload “IS” followed by an Initialization Segment. After the Initialization Segment, the Receiver is expected to send another text message with payload “IS_end” followed by Media Segments. If a new Initialization Segment is received after establishment of the media-delivery WebSocket connection, then the broadcast receiver will send a text message over the same WebSocket connection with the payload “IS” immediately after the last Media Segment associated with the previous Initialization Segment. Then the broadcast receiver will send the new Initialization Segment followed by the text message with payload “IS_end” and then ensuing Media Segments.

8.2.1.2 Media WebSocket Connection Operation

When a Broadcaster Application requests a connection to a media WebSocket, the Receiver shall begin sending the appropriate content once the connection is established. The data sent by the Receiver over a media WebSocket connection is expected to be matched to the type of media indicated by the WebSocket Interface Function as provided in Table 8.1. Therefore, video compliant with formatting described in A/331 [1] shall be sent by the Receiver over the WebSocket identified by the URL *WSPath/atscVid*. Similarly, audio compliant with formatting described in A/331 [1] and captions compliant with formatting described in A/331 [1] shall be sent over *WSPath/atscAud* and *WSPath/atscCap* WebSocket connections, respectively. For media WebSocket connections that are currently open, content may not be sent at all times, for example, if no captions are present at a given time.

8.3 Data Binding

Once the connection is established to the Receiver WebSocket command and control Server, messages can be sent and received. However, since the WebSocket interface is just a plain bidirectional interface with no structure other than message framing, a message format needs to be defined. This section defines the basic formatting of messages, and the following section defines the specific messages that are supported. The message syntax used in this document is defined in the JSON Schema specification [35].

The WebSocket interface for command and control shall be the JSON-RPC 2.0 Specification in Annex E. JSON-RPC provides RPC (remote procedure call) style messaging, including unidirectional notifications and well-defined error handling using the JavaScript Object Notation (JSON) data structure [18].

JSON objects shall be encoded stringified as UTF-8. The Receiver shall parse the JSON object and route the method to the right handler for further processing. Several types of data messages are defined for the command and control WebSocket interface:

- Request message – used to request information or initiate an action
- Synchronous response – a definitive answer to a request provided immediately
- Asynchronous response – a definitive answer to the request provided asynchronously
- Error response – a definitive error to the request provided
- Notification – unidirectional notification, no synchronous or asynchronous response is expected

The other three WebSocket interfaces are used for delivery of binary data from the Receiver to the Broadcaster Application.

The notation used to describe the flow of data in this specification shall be as follows:

```
--> data sent to Receiver  
<-- data sent to Broadcaster Application
```

Note: The interface is bidirectional, so requests, responses and notifications can be initiated by either the Receiver or the Broadcaster Application.

Request/response example:

```
--> { "jsonrpc": "2.0",  
      "method": "exampleMethod1",  
      "params": 1,  
      "id": 1  
}  
  
<-- {  
      "jsonrpc": "2.0",  
      "result": 1,  
      "id": 1  
}
```

Notification example:

```
<-- {  
      "jsonrpc": "2.0",  
      "method": "update",  
      "params": [1, 2, 3, 4, 5]  
}
```

Note that the lack of an 'id' term indicates that no response is expected in the case of a notification.

Error example:

```
--> {  
      "jsonrpc": "2.0",  
      "method": "faultyMethod",  
      "params": 1,  
      "id": 6  
}  
  
<-- {  
      "jsonrpc": "2.0",  
      "error": {"code": -32601, "message": "Method not found"},  
      "id": 6  
}
```

8.3.1 Cancel Request Command

The Cancel Request Command may be used to terminate a selected number or all outstanding JSON RPC requests. The request message supplies a list of IDs corresponding to the request IDs. The response message lists the requests that have been terminated. If no requests matching the requested IDs can be found, an error response is provided. If a request is successfully canceled, the Receiver shall issue a response to that request with an error code indicating that the request was canceled. A cancel request shall not be used to cancel a previous cancel request.

The Cancel Request Command shall be defined as follows:

method: "cancel"

params: A JSON object consisting of a key named `requestIDs` containing a list of request IDs to be canceled.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "requestIDs": {
      "type": "array",
      "items": {"type": "integer"}
    }
  }
}
```

`requestIDs` – This optional parameter shall contain one or more IDs of outstanding requests. If the `requestIDs` list is not supplied, then all outstanding requests shall be canceled.

Response:

result: a JSON object containing the list, `cancelList`, of objects, each object containing the canceled request ID, `requestID`, the disposition of that cancel request, `disposition`, and a description of the disposition, `description`. The Receiver shall accumulate the canceled request IDs so that a single response can be supplied. Note that each request receives a separate error response indicating that the request was canceled.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "cancelList": {"type": "array", "items": {"type": "object",
      "properties": {
        "requestID": {"type": "integer"},
        "disposition": {"type": "string", "enum": ["CANCELED", "UNKNOWN",
"FAILED"]},
        "description": {"type": "string"},
      },
      "required": ["requestID", "disposition"]
    }},
    "required": ["cancelList"]
  }
}
```

`cancelList` – This required property shall provide a list of objects in response to the cancel request. The list shall have the same number of objects as the number of request IDs in the cancel command or the number of outstanding requests if no request IDs were supplied requesting that all outstanding requests be canceled.

`requestID` – This required property shall contain one of the request IDs provided in the cancel command or the `requestID` of a canceled request if all requests were to be canceled.

`disposition` – This required property shall contain one of the following values:

CANCELED – Shall indicate that the request corresponding to the object's `requestID` was successfully canceled.

UNKNOWN – Shall indicate that the request corresponding to the object's `requestID` was not found. The request was not canceled because the Receiver could not identify an outstanding request corresponding to the given `requestID`, or because the request with that ID value was completed before the cancel request was processed.

FAILED – Shall indicate that the request corresponding to the object's `requestID` could not be canceled. In this case, the Receiver found the specified request but could not successfully cancel the request.

`description` – This optional property shall contain a description of the cancel operation.

For example, the Broadcaster Application makes a query request with ID '12' and, due to a user action removing the need for the query, the Broadcaster Application cancels the request with the following command:

```
--> {
  "jsonrpc": "2.0",
  "method": "cancel",
  "params": {
    "requestIDs": [12]
  },
  "id": 913
}
```

The Receiver might respond to the cancel request as follows:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "cancelList": [
      { "requestID": 12,
        "disposition": "CANCELED",
        "description": "Request canceled successfully" }
    ]
  },
  "id": 913
}
```

The Receiver might also issue the following response to query request with ID 12:

```
<-- {
  "jsonrpc": "2.0",
  "error": { "code": -20, "message": "Request Canceled" },
  "id": 12
}
```

As a further example, the Broadcaster Application may wish to cancel all outstanding requests of a function because the user has switched modes of operation. In this case, the Broadcaster Application attempts to cancel the corresponding outstanding requests as follows:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "cancel",  
  "params": {  
    "requestIDs": [42, 216, 922]  
  },  
  "id": 226  
}
```

The Receiver might respond to the cancel request as follows:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {  
    "cancelList": [  
      { "requestID": 42,  
        "disposition": "UNKNOWN",  
        "description": "Request is not currently pending" }  
      { "requestID": 216,  
        "disposition": "CANCELED",  
        "description": "Request canceled successfully" }  
      { "requestID": 922,  
        "disposition": "CANCELED",  
        "description": "Request canceled successfully" }  
    ]  
  },  
  "id": 226  
}
```

The Receiver issues error responses to query requests with IDs 216 and 922. Note that the UNKNOWN disposition for requestID 42 may not be an error since the request could have completed just as the cancel command was issued.

As a further example, the Broadcaster Application wishes to shut down all operations and terminate all outstanding requests. In this case, the Broadcaster Application attempts to cancel all outstanding requests as follows:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "cancel",  
  "id": 226  
}
```

The Receiver might respond to the cancel request as follows:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "cancelList": [
      { "requestID": 324,
        "disposition": "CANCELED",
        "description": "Request canceled successfully" }
      { "requestID": 167,
        "disposition": "CANCELED",
        "description": "Request canceled successfully" }
    ]
  },
  "id": 226
}
```

The Receiver also issues error responses to query requests with IDs 324 and 167. The Broadcaster Application can assume that all outstanding requests have been canceled.

8.3.2 Error Handling

JSON-RPC 2.0 defines a set of reserved error codes. See the table in Annex E Section 5.1.

ATSC-defined error codes from the Receiver shall be as defined in Table 8.2.

Table 8.2 JSON-RPC ATSC Error Codes

Code	Message	Meaning
-1	Unauthorized	Request cannot be honored due to domain restrictions.
-2	Not enough resources	No resources available to honor the request.
-3	System in standby	System is in standby. Request cannot be honored.
-4	Content not found	Requested content cannot be found. For example, invalid URL.
-5	No broadband connection	No broadband connection available to honor the request.
-6	Service not found	The requested Service cannot be located.
-7	Service not authorized	The requested Service was acquired but is not authorized for viewing due to conditional access restrictions.
-8	Video scaling/position failed	The request to scale and/or position the video did not succeed.
-9	XLink cannot be resolved	The request to resolve an XLink has failed.
-10	Track cannot be selected	The media track identified in the Media Track Selection API cannot be found or selected.
-11	The indicated MPD cannot be accessed	In response to the Set RMP URL API, the MPD referenced in the URL provided cannot be accessed.
-12	The content cannot be played	In response to the Set RMP URL API, the requested content cannot be played.
-13	The requested MPD Anchor cannot be reached	In response to the Set RMP URL API, the MPD Anchor indicated cannot be reached (e.g. beyond the end of the file).
-14	Unsupported Content Protection System	The specified content protection system is not supported by the Receiver.
-15	Illegal URL Format	The URL format specified in <code>sourceURL</code> or <code>targetURL</code> of the request is illegal.

-16	Illegal URL Format	The URL format specified in one or more URLs in the requested list is illegal.
-17	Malformed DASH Period	The format of the MPEG DASH fragment specified in the Period is illegal.
-18	MPD not found	The referenced MPD file cannot be found.
-19	The synchronization specified by rmpSyncTime cannot be achieved	In response to the Set RMP URL API with rmpSyncTime, the synchronization indicated by rmpSyncTime cannot be achieved.
-20	Request Canceled	The Broadcaster Application issued the cancel request command (Section 8.3.1) with the ID corresponding to this request.
-21	Changing RMP playback from the current source is not supported	In response to the Set RMP URL API, the Receiver does not support changing playback from the current source to an alternate source (e.g., broadband or locally cached content).
-100	EME TypeError	See EME Section 6.5 [24]
-101	EME NotSupportedError	See EME Section 6.5 [24]
-102	EME InvalidStateError	See EME Section 6.5 [24]
-103	EME QuotaExceededError	See EME Section 6.5 [24]

9. SUPPORTED METHODS

This chapter describes the methods that are supported on the command and control WebSocket Interface. These APIs are based on JSON-RPC 2.0 over WebSockets as described in Section 8. See above chapters for more information on the interface and data binding. All methods are in a reverse domain notation separated with a dot “.”. All ATSC methods that are available over the interface are prefixed with “org.atsc”, leaving room for other methods to be defined for new receiver APIs in the future.

9.1 API Revision Control

To avoid issues with backwards compatibility, the present revision of the standard does not make changes to the semantics of keys within an API but, instead, deprecates the entire API and creates a new API to support the changed functionality. Further, no keys within an API have been deleted and only new keys have been added. For some APIs, values for existing keys may have been added, datatypes for key values may have changed where the types are compatible, and additional enumeration values may have been defined. The Broadcaster Application is expected to gracefully ignore unknown keys and unknown values for existing keys, including unknown enumeration values.

Table 9.1 lists the APIs and groups of APIs and indicates whether they are applicable to AMP operation, RMP operation, or both. The Changes column of Table 9.1 provides a notice that the associated API has changed in a revision of this standard. The following list provides the possible entries for the Changes column and their meaning.

No Entry – The API has no modifications in this revision.

Added – A new API has been defined in this revision.

Deprecated – The associated API has been deprecated. An alternate API is referenced in the Reference column. If an API has been deprecated, the Receiver may continue to support the API, but the Broadcaster Application cannot rely on its availability in a future revision and should migrate to the referenced API. Deprecated APIs can be found in Annex C (A/344:2019) and Annex D (A/344:2020).

Description Changed – The text supporting the description of the API has changed. No substantive changes have been made to the semantics or syntax of the API.

Extended – The API has been extended by adding new keys or values. Previous keys and values remain as previously defined. A description of the changes made or copy of the previous API can be found in Annex C (A/344:2019) or Annex D (A/344:2020).

Table 9.1 API Applicability

WebSocket APIs	Reference	Applicability	Changes
Cancel Request Command	Section 8.3.1	Always	Added in A/344:2020
Receiver Query APIs	Section 9.2	Always	Extended in A/344:2019
Query MPD URL API	Use Section 9.2.10	RMP	Deprecated in A/344:2020
Query Alerting URL API	Section 9.2.8	Always	Deprecated in A/344:2019
Query Alerting Signaling API	Section 9.2.8	Always	Added in A/344:2019
Query Service Guide URLs API	Section 9.2.9	Always	Added in A/344:2020
Query Signaling Data API	Section 9.2.10	Always	Added in A/344:2020
MPD Change Notification API	Use Section 9.3.11	RMP	Deprecated in A/344:2020
Alerting Change Notification API	Section 9.3.8	Always	Deprecated in A/344:2019 [†]
Alerting Change Notification API	Section 9.3.8	Always	Added in A/344:2019 [†]
Service Guide Change Notification API	Section 9.3.10	Always	Added in A/344:2020
Signaling Data Change Notification API	Section 9.3.11	Always	Added in A/344:2020
Cache Request APIs	Section 9.4	Always	
Query Cache Usage API	Section 9.5	Always	
Event Stream APIs	Section 9.6	RMP	
Acquire Service API	Section 9.7.1	Always	
Video Scaling and Positioning API	Section 9.7.2	RMP	Extended in A/344:2019
XLink Resolution API	Use Sections 9.7.5 and 9.16	RMP	Deprecated in A/344:2019
Subscribe MPD Changes API	Use Section 9.7.5.1	AMP	Deprecated in A/344:2019
Unsubscribe MPD Changes API	Use Section 9.7.5.2	AMP	Deprecated in A/344:2019
Set RMP URL API	Section 9.7.3	RMP	Extended in A/344:2019
Audio Volume API	Section 9.7.4	RMP	
Integrated Subscribe API	Section 9.7.5.1	Always	Added in A/344:2019, Extended in A/344:2020
Integrated Unsubscribe API	Section 9.7.5.2	Always	Added in A/344:2019, Extended in A/344:2020
Subscribe Alerting Changes	Use Section 9.7.5.1	Always	Deprecated in A/344:2019
Unsubscribe Alerting Changes	Use Section 9.7.5.2	Always	Deprecated in A/344:2019
Subscribe Content Changes	Use Section 9.7.5.1	Always	Deprecated in A/344:2019
Unsubscribe Content Changes	Use Section 9.7.5.2	Always	Deprecated in A/344:2019
Subscribe RMP Media Time Change Notification	Use Section 9.7.5.1	RMP	Deprecated in A/344:2019
Unsubscribe RMP Media Time Change Notification	Use Section 9.7.5.2	RMP	Deprecated in A/344:2019
Subscribe RMP Playback State Change Notification	Use Section 9.7.5.1	RMP	Deprecated in A/344:2019
Unsubscribe RMP Playback State Change Notification	Use Section 9.7.5.2	RMP	Deprecated in A/344:2019
Subscribe RMP Playback Rate Change Notification	Use Section 9.7.5.1	RMP	Deprecated in A/344:2019

Unsubscribe RMP Playback Rate Change Notification	Use Section 9.7.5.2	RMP	Deprecated in A/344:2019
Media Track Selection API	Section 9.8	RMP	
Mark Unused API	Section 9.9	Always	Description changed in A/344:2019
Content Recovery APIs	Section 9.10	Always	
Filter Code APIs	Section 9.11	Always	
Keys APIs	Section 9.12	Always	
Query Device Info API	Section 9.13	Always	Extended in A/344:2019 and A/344:2020
RMP Content Synchronization APIs	Section 9.14	RMP	
Digital Rights Management (DRM) APIs	Section 9.15	Always	Added in A/344:2019
XLink Management APIs	Section 9.16	RMP	Added in A/344:2019

† Note that the Alerting Change Notification API was substantially changed in A/344:2019 resulting in the previous API definition being deprecated with a new API defined with the same name. Essentially, the API was modified to pass the alerting XML fragment(s) instead of a URL reference to the alerting XML fragment(s). The previous API definition can be found in Annex C Section C.2.

9.2 Receiver Query APIs

The Receiver software stack exposes a set of WebSocket APIs to the Broadcaster Application to retrieve user settings and information, as described in the following sections.

If these settings are not available from the Receiver, the Broadcaster Application may use default values based on its own business policy and logic. A Broadcaster Application may choose to provide its own settings user interface and store the collected settings as cookies on the Receiver.

The following settings and information may be retrieved by a Broadcaster Application:

- Content Advisory Rating setting
- State of Closed Caption display (enabled/disabled)
- Current Service ID
- Language preferences (Audio, User Interface, Captions, etc.)
- Closed Caption Display Preferences (font sizes, styles, colors, etc.)
- Audio Accessibility preferences
- A URL the Broadcaster Application can use to fetch the current broadcast MPD
- Receiver Web Server URI to access the corresponding Application Context Cache
- XLink Period

The following APIs are defined to allow Broadcaster Applications to retrieve these settings and information.

9.2.1 Query Content Advisory Rating API

Broadcaster Applications may wish to know the highest content advisory rating the viewer has unlocked on a receiver, in order to decide what applications, links or text within a page to make available to the user. For example, the Broadcaster Application can use this rating to decide whether a description of a program should be presented to the viewer. If the rating is changed on the Receiver, the Receiver shall send an event notification to the Broadcaster Application,

indicating so, if the Receiver supports this setting and the Broadcaster Application has subscribed to the notification (Section 9.7.5). If the Receiver does not make this information available, the Broadcaster Application may choose a default value based on its own business logic and policy.

Furthermore, the Broadcaster Application may wish to discover the current content advisory rating as signaled in the content currently being rendered by the Receiver and whether that content is being blocked or not.

The Query Content Advisory Rating API shall be defined as follows:

method: "org.atsc.query.ratingLevel"

params: Omitted

Response:

result: a JSON object containing a `rating` key/value pair, a `contentRating` key/value pair and a `blocked` key/value pair.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "rating": {"type": "string"},
    "contentRating": {"type": "string"},
    "blocked": {"type": "boolean"}
  },
  "required": ["rating", "contentRating", "blocked"]
}
```

`rating` – This required string shall provide the content advisory rating in string format, as defined in A/331 [1], Section 7.3.

`contentRating` – This required string value shall provide the content advisory rating of the content currently being rendered by the Receiver Media Player in string format, as defined in A/331 [1], Section 7.3.

`blocked` – This required Boolean value shall indicate whether the Receiver is currently blocking the content due to the content advisory rating of the service being higher than the content advisory rating preference.

For example, consider the case that the content advisory rating setting is TV-PG-D-L for the US Rating Region 1 and the current content advisory rating is TV-G. The Broadcaster Application can make a request:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.ratingLevel",
  "id": 37
}
```

The Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "rating": "1, 'TV-PG-D-L', {0 'TV PG'}{1 'D'}{2 'L'}",
    "contentRating": "1, 'TV-G', {0 'TV G'}",
    "blocked": false
  },
  "id": 37
}
```

9.2.2 Query Closed Captions Enabled/Disabled API

The Broadcaster Application may wish to know whether the user has turned on closed captions, in order to display its application graphics at a different location than where the closed captions are typically presented. The Broadcaster Application requests the closed caption setting from the Receiver via Receiver WebSocket Server interface.

The Receiver sends an event when the user enables or disables closed captioning to the running Broadcaster Application.

The Query Closed Captions Enabled/Disabled API shall be defined as follows:

method: "org.atsc.query.cc"

params: Omitted

Response:

result: a JSON object containing a `ccEnabled` key/value pair.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "ccEnabled": {"type": "boolean"}
  },
  "required": ["ccEnabled"]
}
```

`ccEnabled` – This required Boolean shall indicate true if closed captions are currently enabled by the user and false otherwise.

For example, if closed captions are currently enabled:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.cc",
  "id": 49
}
```

The Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"ccEnabled": true},
  "id": 49
}
```

9.2.3 Query Service ID API

Since the same application may be used for multiple services within the same broadcast family, the Broadcaster Application may wish to know the currently selected Service. This allows the Broadcaster Application to adjust its user interface and provide additional features that might be available on one service vs. another.

The Query Service ID API shall be defined as follows:

method: "org.atsc.query.service"

params: Omitted

Response:

result: a JSON object containing a `service` key/value pair.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "service": {"type": "string", "format": "uri"}
  },
  "required": ["service"]
}
```

`service` – This required key shall indicate the globally unique Service ID associated with the currently selected service as given in the SLT in **SLT.Service@globalServiceID**. See A/331 [1] Section 6.3.

For example, if the globally unique Service ID for the currently selected services is "https://doi.org/10.5239/8A23-2B0B", and the Broadcaster Application issues a request to the Receiver:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.service",
  "id": 55
}
```

The Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"service": "https://doi.org/10.5239/8A23-2B0B"},
  "id": 55
}
```

9.2.4 Query Language Preferences API

The Broadcaster Application may wish to know the language settings in the Receiver, including the language selected for audio output, user interface displays, and subtitles/captions. The Broadcaster Application may use the Query Language Preferences API to determine these settings.

The Query Language Preferences API shall be defined as follows:

method: "org.atsc.query.languages"

params: Omitted

Response:

result: a JSON object containing an object with three key/value pairs as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "preferredAudioLang": {"type": "string"},
    "preferredUiLang": {"type": "string"},
    "preferredCaptionSubtitleLang": {"type": "string"}
  }
}
```

preferredAudioLang, preferredUiLang, preferredCaptionSubtitleLang – Each of these strings indicates the currently set language preference of the respective item, coded according to RFC 5646 [17].

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.languages",
  "id": 95
}
```

Moreover, if the user lives in the U.S. but has set his or her language preference for audio tracks and caption/subtitles to Spanish, the Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "preferredAudioLang": "es",
    "preferredUiLang": "en",
    "preferredCaptionSubtitleLang": "es"
  },
  "id": 95
}
```

9.2.5 Query Caption Display Preferences API

The Broadcaster Application may wish to know the user's preferences for closed caption displays, including font selection, color, opacity and size, background color and opacity, and other characteristics. The Broadcaster Application may use the Query Caption Display Preferences API to determine these settings.

The Query Caption Display Preferences API shall be defined as follows:

method: "org.atsc.query.captionDisplay"

params: Omitted

Response:

result: a JSON object containing an object with key/value pairs as defined below.

result JSON Schema:


```

{
  "type": "object",
  "properties": {
    "msgType": {"enum": ["captionDisplayPrefs"]},
    "cta708": {"type": "object",
      "properties": {
        "characterColor": {"type": "string"},
        "characterOpacity": {"type": "number"},
        "characterSize": {"type": "integer"},
        "fontStyle": {"enum": [
          "Default",
          "MonospacedSerifs",
          "ProportionalSerifs",
          "MonospacedNoSerifs",
          "ProportionalNoSerifs",
          "Casual",
          "Cursive",
          "SmallCaps"
        ]},
        "backgroundColor": {"type": "string"},
        "backgroundOpacity": {"type": "number"},
        "characterEdge": {"enum": [
          "None",
          "Raised",
          "Depressed",
          "Uniform",
          "LeftDropShadow",
          "RightDropShadow"
        ]},
        "characterEdgeColor": {"type": "string"},
        "windowColor": {"type": "string"},
        "windowOpacity": {"type": "number"}
      }
    },
    "imsc1": {"type": "object",
      "properties": {
        Key / value pairs as described in Section 9.3.6.1
      }
    }
  },
  "required": ["msgType"],
  "anyOf": ["cta708", "imsc1"]
}

```

The semantics for the key/value pairs in the "cta708" object shall be the same as for the Caption Display Preferences Change Notification API in Section 9.3.6. In addition, IMSC1 [36] attributes as specified in Section 9.3.6.1 may be included.

For example, if the Broadcaster Application makes a query:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.query.captionDisplay",  
  "id": 932  
}
```

The Receiver might respond:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {  
    "msgType": "captionDisplayPrefs",  
    "cta708": {  
      "characterColor": "#F00000",  
      "characterOpacity": 0.5,  
      "characterSize": 80,  
      "fontStyle": "MonospacedNoSerifs",  
      "backgroundColor": "#808080",  
      "backgroundOpacity": 0,  
      "characterEdge": "None",  
      "characterEdgeColor": "#000000",  
      "windowColor": "#000000",  
      "windowOpacity": 0  
    },  
    "imsc1": {  
      "region_textAlign": "center",  
      "content_fontWeight": "bold"  
    }  
  },  
  "id": 932  
}
```

9.2.6 Query Audio Accessibility Preferences API

The Broadcaster Application may wish to know the audio accessibility settings in the Receiver, including whether the automatic rendering of the following is enabled: video description service, audio/aural representation of emergency information and what are the corresponding language preferences. The Broadcaster Application may use the Query Audio Accessibility Preferences API to determine these settings.

The Query Audio Accessibility Preferences API shall be defined as follows:

method: "org.atsc.query.audioAccessibilityPref"

params: Omitted

Response:

result: a JSON object containing an object as defined below.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "videoDescriptionService": {
      "type": "object",
      "properties": {
        "enabled": {"type": "boolean"},
        "language": {"type": "string"}
      }
    },
    "audioEIService": {
      "type": "object",
      "properties": {
        "enabled": {"type": "boolean"},
        "language": {"type": "string"}
      }
    }
  }
}

```

`videoDescriptionService.enabled`, `audioEI.enabled` – Each of these Boolean values respectively shall indicate the current state of automatic rendering preference of video description service (VDS), audio/aural representation of emergency information.

`videoDescriptionService.language` – A string that shall indicate the preferred language of VDS rendering, coded according to RFC 5646 [17].

`audioEI.language` – A string that shall indicate the preferred language of audio/aural representation of emergency information rendering, coded according to RFC 5646 [17].

When a Receiver does not have a setting for `videoDescriptionService.enabled`, `videoDescriptionService.language`, `audioEI.enabled`, `audioEI.language` then it is expected that the response does not include the corresponding property.

For example, the Broadcaster Application makes a query:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.audioAccessibilityPref",
  "id": 90
}

```

In addition, if the user has set his or her automatic rendering preference setting of video description service set to ON and the Receiver does not have the rest of the settings, then the Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "videoDescriptionService": {
      "enabled": true
    }
  },
  "id": 90
}
```

9.2.7 Query Receiver Web Server URI API

The Broadcaster Application may wish to access the location of the Application Context Cache provided by the Receiver. This conceptual cache provides access to resources delivered under the auspices of the Application Context Identifier defined for the currently loaded Broadcaster Application. These are made available through the Receiver Web Server using a Base URI (see Section 5.3). This API provides access to that URI.

The Receiver Web Server URI API shall be defined as follows:

method: "org.atsc.query.baseURI"

params: Omitted

Response:

result: a JSON object containing an object as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "baseURI": {"type": "string", "format": "uri"}
  },
  "required": ["baseURI"]
}
```

baseURI – This return parameter shall contain the URI where the resources associated with the Application Context Identifier may be accessed.

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.baseURI",
  "id": 90
}
```

The Receiver responds with the URI of the Receiver Web Server for the Application Context Cache defined for the current Application Context Identifier:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "baseURI": "http://localhost:8080/contextA"
  },
  "id": 90
}
```

The resulting URI can be prepended to relative references to resources to access those resources on the Receiver.

9.2.8 Query Alerting Signaling API

The Broadcaster Application may wish to access the various alerting metadata structures signaled in the current broadcast. The Query Alerting Signaling API returns a list of the specific alerting metadata the Broadcaster Application has requested.

The Query Alerting Signaling API shall be defined as follows:

method: "org.atsc.query.alerting"

params: A JSON object consisting of a key named `alertingTypes` with a list of enumerated values. An empty list is equivalent to supplying all values.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "alertingTypes": {
      "type": "array",
      "items": {
        "type": "string", "enum": ["AEAT", "OSN"]
      }
    }
  },
  "required": ["alertingTypes"]
}
```

`alertingTypes` – An array of one or both of the alerting types as follows:

AEAT – Requests the most recent AEAT XML fragment, if any.

OSN – Requests the most recent OSN XML fragment, if any.

Response:

result: a JSON object containing a list of XML fragments as defined below.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "alertList": { "type": "array", "items": { "type": "object",
      "properties": {
        "alertingType": { "type": "string", "enum": ["AEAT", "OSN"] },
        "alertingFragment": { "type": "string", "format": "xml" },
        "receiveTime": { "type": "string", "format": "xs:dateTime" },
        "filteredEventList": {
          "type": "array",
          "items": {
            "aeald": { "type": "string" }
          }
        }
      }
    }
  },
  "required": ["alertingType", "alertingFragment"],
  "allOf": [
    {
      "if": {
        "properties": {
          "alertingType": { "enum": ["OSN"] }
        }
      },
      "then": { "required": ["receiveTime"] }
    }
  ]
},
{
  "required": ["alertList"]
}

```

alertingType – This required parameter shall contain one of the alerting types, "AEAT" or "OSN".

The corresponding **alertingFragment** shall contain the data corresponding to the type of alerting metadata fragment indicated.

alertingFragment – This required string shall contain the alerting XML fragment for the associated **alertingType**. The AEAT XML and OSN XML fragments are extracted from their respective LLS tables that are described in A/331 [1].

receiveTime – The date and time when the alerting fragment was received. This value shall be provided when the object is "OSN". (Note: The **onscreenMessageNotification** element includes a **keepScreenClear@notificationDuration** attribute which is the duration of the **KeepScreenClear** message starting from the time the OSN was received. Thus, the time the OSN was received is necessary for the Broadcaster Application to fully utilize the OSN information.) The **xs:dateTime** XML data type shall be formatted as defined in the W3C XML Schema [28].

filteredEventList – Provides a list of AEA events that have been filtered by the Receiver. The Receiver may elect to filter an event for a variety of reasons based on user preferences, location or some other criteria. If an AEA event is filtered, the corresponding **AEAT.AEA@aeaId** shall appear in the **filteredEventList** property. If an AEA event is not filtered, the corresponding **AEAT.AEA@aeaId** shall not appear in the list. An empty or absent **filteredEventList** indicates

that no events have been filtered by the Receiver. This property is only applicable when the `alertingType` is "AEAT".

`aeaId` – The `AEAT.AEA@aeaId` of an AEA event that has been filtered by the Receiver. See the previous description of the `filteredEventList` property for the usage of this property.

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.alerting",
  "params": {
    "alertingTypes": ["AEAT", "OSN"]
  },
  "id": 913
}
```

The Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "alertList": [
      { "alertingType": "AEAT",
        "alertingFragment": "<AEAT>...</AEAT>" },
      { "alertingType": "OSN",
        "alertingFragment": "<OSN>...</OSN>",
        "receiveTime": "2017-01-01T23:54:59.590Z" }
    ]
  },
  "id": 913
}
```

9.2.9 Query Service Guide URLs API

The Broadcaster Application may wish to access the various service guide data structures provided in the current broadcast. The Query Service Guide URLs API returns a list of URLs the Broadcaster Application can use to retrieve (for example, by XHR) the specific service guide data structures provided in the broadcast.

The Query Service Guide URLs API shall be defined as follows:

method: "org.atsc.query.serviceGuideUrls"

params: A JSON object consisting of an optional key named `service`.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "service": { "type": "string", "format": "url" }
  }
}
```

`service` – The optional `service` field as defined in the Query Service ID API in Section 9.2.3.

When omitted, all service guide fragments are returned for all services. When present, only those fragments related to the provided service are returned.

Response:

result: a JSON object containing a list of references as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "urlList": { "type": "array", "items": { "type": "object",
      "properties": {
        "sgType": { "type": "string", "enum": ["Service", "Schedule", "Content"] },
        "sgUrl": { "type": "string", "format": "url" },
        "service": { "type": "string", "format": "url" },
        "content": { "type": "string", "format": "url" }
      },
      "required": ["sgType", "sgUrl"]
    }
  },
  "required": ["urlList"]
}
```

sgType – One of the service guide XML fragment types. The corresponding **sgUrl** can be used to access the XML fragment corresponding to the type of service guide fragment indicated. Note that there may be multiple fragments within the list of the same **sgType**. The **sgType** may be used to quickly access fragments of interest.

sgUrl – A fully-qualified URL that can be used by the Broadcaster Application, for example in an XHR request, to retrieve the current broadcast service guide XML fragment for the associated **sgType**. The service guide is delivered in XML fragments whose syntax is defined in A/332 [2].

service – The optional **service** field as defined in the Query Service ID API in Section 9.2.3. Note: This is more commonly known as the **globalServiceId** field. For proper operation this requires that **globalServiceId** be present in the SLT. See A/331 [1] Section 6.3 and A/351 [30] Section 5.

content – A fully-qualified URI present when **sgType**="content" that can be used by the Broadcaster Application to uniquely identify a specific content item on a service (of which there may be many). The service guide is delivered in XML fragments whose syntax is defined in A/332 [2].

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.serviceGuideUrls",
  "id": 913
}
```

The Receiver might respond:


```

<-- {
  "jsonrpc": "2.0",
  "result": {
    "urlList": [
      { "sgType": "Service",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Service.xml",
        "service": "https://doi.org/10.5239/8A23-2B0B" },
      { "sgType": "Schedule",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Schedule.xml",
        "service": "https://doi.org/10.5239/8A23-2B0B" },
      { "sgType": "Content",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Content.xml",
        "service": "https://doi.org/10.5239/8A23-2B0B",
        "content": "urn:eidr:10.5240:7791-8534-2C23-9030-8610-5" }
    ],
    "id": 913
  }
}

```

Note that the URLs provided are examples only. The actual URLs used, including the file names, are completely dependent on the Receiver implementation and how it chooses to make the ESG files available through its HTTP server. The Broadcaster Application should make no assumptions regarding the URL path and simply use it to access the fragment data directly.

The referenced service guide files, in this example, `Service.xml`, `Schedule.xml` and `Content.xml`, shall contain the Service, Schedule and Content XML fragments as described in A/332 [2], respectively. The Receiver shall extract each XML fragment from the binary SGDU structure before making it available to the Broadcaster Application.

To associate ESG files with Broadcaster Applications, the corresponding Application Context Identifiers shall be provided in the Extended FDT (EFDT) element, **FDT-Instance@appContextIdList**, defined when sending the ESG files in the LCT channel of the ESG Service ROUTE session. Descriptions of the FDT extensions and the ESG Service can be found in A/331 [1]. Application Context Identifiers need not be included in the EFDT if the ESG data is not needed by the Broadcaster Application.

9.2.10 Query Signaling Data API

The Broadcaster Application may wish to access the various signaling metadata structures from the current broadcast. The Query Signaling API returns a list of signaling tables that the Broadcaster Application can use to extract details not otherwise available such as major and minor channel numbers.

The Query Signaling API shall be defined as follows:

method: "org.atsc.query.signaling"

params: A JSON object consisting of a key named `objectNames` with a list of enumerated values.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "nameList": {
      "type": "array",
      "items": { "type": "string" }
    },
    "required": ["nameList"]
  }
}

```

nameList – An array of object names as described below in **names**. If absent, no tables or Metadata Object Types are returned.

names – This field shall be set to an integrated list of **LLS_table_id** values and signaling metadata **object_type** values as enumerated in A/331, Table 6.17, “Metadata Object Types” [1]. The **LLS_table_id** values shall be encoded as a decimal integer (i.e. not hexadecimal). When the **names** field is omitted, this request shall return all of the supported current LLS tables and all of the supported current Metadata Object Types. Note: Some Metadata Object Types are transport dependent (ROUTE versus MMT) and might not be available on a given Receiver model. A request to return the LLS SignedMultiTable, ALL, AEI, EMSG, and EVTI is not expected to return any LLS tables or Metadata Object Types; that is no such LLS tables and Metadata Object Types as well as any unsupported ones would be returned. Note that LLS tables can be delivered via the SignedMultiTable.

Response:

result: a JSON object containing a list of tables as text strings as defined below.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "objectList": { "type": "array", "items": { "type": "object",
      "properties": {
        "name": { "type": "string" },
        "group": { "type": "integer" },
        "version": { "type": "integer" },
        "table": { "type": "string", "format": "xml" }
      },
      "required": ["name", "version", "table"]
    } },
    "required": ["objectList"]
  }
}

```

name – See **names** above.

version – This is the version of the XML signaling document. For LLS, it shall be set to the value of **LLS_table_version**. For Metadata Object Types, it shall be set to **metadataEnvelope@version**.

group – For LLS tables, this is required and shall be the value of `LLS_group_id`. For Metadata Object Types this shall be omitted.

table – This string shall contain a single object of a type matching `name`, encoded in UTF-8. For objects that are not strings, they shall first be encoded as Base64. The XML documents shall be uncompressed. For both LLS and SLS documents, only the XML document itself is returned. Related packaging including signatures, binary LLS fields (e.g. `group_count_minus1`), MIME separators, etc. shall be removed.

For example, the application makes a query for the SLT and MPD as follows:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.signaling",
  "params": {
    "objectList": ["1", "MPD"]
  },
  "id": 913
}
```

The Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "objectList": [
      { "name": "1",
        "group": "1",
        "version": "23",
        "table": "<SLT ..... </SLT>" },
      { "name": "MPD",
        "version": "65",
        "table": "<MPD ..... </MPD>" }
    ]
  },
  "id": 913
}
```

9.3 Asynchronous Notifications of Changes

The types of notifications that the Receiver shall provide to the Broadcaster Application through the APIs defined in this section are as specified in Table 9.2. All use a `method` of `org.atsc.notify` and include a parameter called `"msgType"` to indicate the type of notification.

Table 9.2 Asynchronous Notifications

msgType	Event Description	Reference
ratingChange	Content Advisory Rating Settings Change – a notification that is provided whenever the user changes the Content Advisory Rating settings in the Receiver.	Sec. 9.3.1
ratingBlock	Content Advisory Rating Block Change – a notification that is provided whenever the user changes the content advisory rating settings in the Receiver such that the currently decoding program goes from blocked to unblocked or unblocked to blocked.	Sec. 9.3.2

serviceChange	Service Change – a notification that is provided if a different service is acquired due to user action, and the new service signals the URL of the same application.	Sec. 9.3.3
captionState	Caption State – a notification that is provided whenever the user changes the state of closed caption display (either off to on, or on to off).	Sec. 9.3.4
langPref	Language Preference – a notification that is provided whenever the user changes the preferred language.	Sec. 9.3.5
captionDisplayPrefs	Closed Caption display properties preferences.	Sec. 9.3.6
audioAccessibilityPref	Audio Accessibilities preferences.	Sec. 9.3.7
alertingChange	Alerting Change – a notification that a new version of either the AEAT or OSN messages has been received or if alert filtering preferences have been changed resulting in events becoming unfiltered.	Sec. 9.3.8
contentChange	Content Change – a notification that new content has been placed in the Application Context Cache and may be accessed by the Broadcaster Application.	Sec. 9.3.9
serviceGuideChange	Service Guide Change – a notification that is provided when new ESG fragments have been received.	Sec. 9.3.10
signalingData	Signaling Data Change – a notification that some new signaling data has been received.	Sec. 9.3.11
contentRecoveryStateChange	Content Recovery State Change – a notification that is provided whenever use of audio watermark, video watermark, audio fingerprint, and/or video fingerprint for content recovery changes.	Sec. 9.10.4
displayOverrideChange	Display Override Change – a notification that is provided if the display override state or the state of blocked application access to certain resources changes.	Sec. 9.10.5
recoveredComponentInfoChange	Recovered Component Info Change – a notification that is provided if a component of the service being received by the Receiver changes at the upstream.	Sec. 9.10.6
rmpMediaTimeChange	RMP Media Time Change – a notification that is provided periodically during playback.	Sec. 9.14.5
rmpPlaybackStateChange	RMP Playback State Change – a notification that is provided if the playback state changes.	Sec. 9.14.6
rmpPlaybackRateChange	RMP Playback Rate Change – a notification that is provided if playback speed changes.	Sec. 9.14.7
xlinkResolution	Xlink Resolution – a notification that is provided when the RMP encounters a period with an XLink attribute.	Sec. 9.16.1

9.3.1 Content Advisory Rating Change Notification API

The Content Advisory Rating Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the user makes any change to the content advisory rating settings in the Receiver.

The Content Advisory Rating Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "ratingChange" and a key/value pair named "rating" representing the new content advisory rating setting.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["ratingChange"]},
    "rating": {"type": "string"}
  },
  "required": ["msgType", "rating"]
}

```

The `rating` string shall conform to the encoding specified in A/331 [1], Section 7.3.

As an example, if the user changes the content advisory rating settings to "TV-PG-D" in the US system (Rating Region 1), then the Receiver would issue a notification to the Broadcaster Application with the new rating level as follows:

```

<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "ratingChange",
    "rating": "{1,'TV-PG-D', {0 'TV PG'}{1 'D'}}"
  },
}

```

9.3.2 Content Advisory Rating Block Change Notification API

The Content Advisory Rating Block Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the user makes a change to the content advisory rating settings in the Receiver that results in a change to the content advisory rating blocking of the currently displayed service, either from unblocked to blocked or vice versa.

When the service is blocked, the Broadcaster Application is expected to remove any potentially objectionable material. Optionally, the Broadcaster Application may offer to tune to alternative content more appropriate to the current content advisory rating settings. When the service is unblocked, the Broadcaster Application is expected to resume normal operations.

In addition to the blocked status, the Receiver might also provide the content advisory rating of the currently displayed service to allow the Broadcaster Application to determine why and perhaps inform the user why the content has been blocked. Note that this notification shall not be issued if there is no change to the blocked status even when the content advisory rating may change.

Note that Content Advisory Ratings, downloadable rating region table(s) and parental control may be addressed by law or regulation.

The Content Advisory Rating Block Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "ratingBlock", a key named "blocked" with a Boolean value representing the state of blocking after the user action or change to the current content, and an optional key named "contentRating" containing the content advisory rating of the currently displayed service.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["ratingBlock"]},
    "blocked": {"type": "boolean"},
    "contentRating": {"type": "string"}
  },
  "required": ["msgType", "blocked"]
}

```

The `contentRating` string shall conform to the encoding specified in A/331 [1], Section 7.3. An example in which the state of program blocking has gone from unblocked to blocked:

```

<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "ratingBlock",
    "blocked": true,
    "contentRating": "1, 'TV-PG-L', {0 'TV PG'}{1 'L'}"
  }
},

```

9.3.3 Service Change Notification API

The Service Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the user changes to another service also associated with the same Broadcaster Application. Note that if the user changes to another service not associated with the same Broadcaster Application, or the new service has no signaled Broadcaster Application, no `serviceChange` event is fired.

The Service Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "serviceChange" and a key named "service" with a string indicating the URI of the new service.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["serviceChange"]},
    "service": {"type": "string"}
  },
  "required": ["msgType", "service"]
}

```

The `service` string shall consist of the globally unique Service ID associated with the newly selected service as given in **SLT.Service@globalServiceID**. See A/331 [1] Section 6.3.

In the following example, the user has caused a service change to a service with a globally unique Service ID "https://doi.org/10.5239/8A23-2B0B":

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "serviceChange",
    "service": "https://doi.org/10.5239/8A23-2B0B"
  }
}
```

9.3.4 Caption State Change Notification API

The Caption State Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the user turns captions on or off.

The Caption State Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "captionState" and a key named "captionDisplay" with a Boolean value representing the new state of closed caption display.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["captionState"]},
    "captionDisplay": {"type": "boolean"}
  },
  "required": ["msgType", "captionDisplay"]
}
```

For example, the Receiver notifies the Broadcaster Application that caption display has been turned on:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "captionState",
    "captionDisplay": true
  }
}
```

9.3.5 Language Preference Change Notification API

The Language Preference Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the user changes the preferred language applicable to either audio, user interfaces, or subtitles/captions.

The Language Preference Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "langPref" and one or more key/value pairs described below.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["langPref"]},
    "preferredAudioLang": {"type": "string"},
    "preferredUiLang": {"type": "string"},
    "preferredCaptionSubtitleLang": {"type": "string"}
  },
  "required": ["msgType"]
}

```

`preferredAudioLang`, `preferredUiLang`, `preferredCaptionSubtitleLang` – Each of these strings indicate the preferred language of the respective item, coded according to RFC 5646 [17]. At least one key/value pair shall be present.

For example, if the user has changed the preferred language of the captions to French as spoken in Canada:

```

<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "langPref",
    "preferredCaptionSubtitleLang": "fr-CA"
  }
}

```

9.3.6 Caption Display Preferences Change Notification API

The Caption Display Preferences Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if there are changes in preferences for display of closed captioning. The API syntax shall be as defined in the JSON schema specified in this section with the addition of IMSC1 [36] attributes (as defined in A/343 [5]) as specified in Section 9.3.6.1. The IMSC1 attributes are defined by text in Section 9.3.6.1 but are intended to be an integral part of the JSON Schema in this section within the `displayPrefs` object.

The Caption Display Preferences Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "captionDisplayPrefs" and a number of keys describing various aspects of the preferences, as defined below.

params JSON Schema:


```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["captionDisplayPrefs"]},
    "cta708": {"type": "object",
      "properties": {
        "characterColor": {"type": "string"},
        "characterOpacity": {"type": "number"},
        "characterSize": {"type": "integer"},
        "fontStyle": {"type": "string", "enum": [
          "Default",
          "MonospacedSerifs",
          "ProportionalSerifs",
          "MonospacedNoSerifs",
          "ProportionalNoSerifs",
          "Casual",
          "Cursive",
          "SmallCaps"
        ]},
        "backgroundColor": {"type": "string"},
        "backgroundOpacity": {"type": "number"},
        "characterEdge": {"type": "string", "enum": [
          "None",
          "Raised",
          "Depressed",
          "Uniform",
          "LeftDropShadow",
          "RightDropShadow"
        ]},
        "characterEdgeColor": {"type": "string"},
        "windowColor": {"type": "string"},
        "windowOpacity": {"type": "number"}
      }
    },
    "imsc1": {"type": "object",
      "properties": {
        Key / value pairs as described in Section 9.3.6.1
      }
    }
  },
  "required": ["msgType"],
  "anyOf": ["cta708", "imsc1"]
}

```

One or more of the caption scheme objects defining the closed caption display preferences may be included in the notification message. The "cta708" object, if present, shall provide one or more of the closed caption display preferences that follow.

characterColor – This parameter shall be a string that shall represent the color of the characters.

The color value shall conform to the encoding for color as specified in the W3C

recommendation for CSS3 color [7] using the “#” 24-bit sRGB notation. For example, red is represented as “#FF0000”.

`characterOpacity` – This parameter shall be an integer or fixed-point number in the range 0 to 1 inclusive that shall represent the opacity of the characters. For example, a value of .33 shall mean 33% opaque; a value of 0 shall mean completely transparent.

`characterSize` – This parameter shall be a percentage multiplier of the default font size where 100 has no change, 50 is ½ size and 200 is double the size. The syntax and semantics shall be consistent with IMSC1 as defined in A/343 [5].

`fontStyle` – This string shall indicate the style of the preferred caption font. The eight possible choices shall be as specified in CTA-708 [31] Section 8.5.3:

- “Default” (undefined)
- “MonospacedSerifs” – Monospaced with serifs (similar to Courier)
- “ProportionalSerifs” – Proportionally spaced with serifs (similar to Times New Roman)
- “MonospacedNoSerifs” – Monospaced without serifs (similar to Helvetica Monospaced)
- “ProportionalNoSerifs” – Proportionally spaced without serifs (similar to Arial and Swiss)
- “Casual” – Casual font type (similar to Dom and Impress)
- “Cursive” – Cursive font type (similar to Coronet and Marigold)
- “SmallCaps” – Small capitals (similar to Engravers Gothic)

`backgroundColor` – This parameter shall represent the color of the character background, given in the same CSS-compatible format as `characterColor`.

`backgroundOpacity` – This parameter shall be an integer or fixed-point number in the range 0 to 1 that shall represent the opacity of the character background. A value of 1 shall mean 100% opaque; a value of 0 shall mean completely transparent.

`characterEdge` – This parameter shall indicate the preferred display format for character edges. The preferred color of the edges (or outlines) of the characters shall be as given in `characterEdgeColor`. Edge opacities shall have the same attribute as the character foreground opacities. The choices and their effects on the character display shall be as specified in CTA-708 [31] Section 8.5.8: “None”, “Raised”, “Depressed”, “Uniform”, “LeftDropShadow”, and “RightDropShadow”.

`characterEdgeColor` – This parameter shall represent the color of the character edges, if applicable, given in the same format as `characterColor`.

`windowColor` – This parameter represents the color of the caption window background, given in the same format as `characterColor`.

`windowOpacity` – This parameter shall be a number in the range 0 to 1 that shall represent the opacity of the caption window. A value of 1 shall mean 100% opaque; a value of 0 shall mean completely transparent.

9.3.6.1 IMSC1 Extensions

The key/value pairs defined in this section provide a means to represent any IMSC1 (as defined in A/343 [5]) attribute preference within the “imsc1” object, if present.

The key/value pairs for IMSC1 preferences shall take the form:

```
"<imsc1_key>": "<imsc1_value>"
```

The syntax of the `<imsc1_key>` shall be as specified below using the Augmented Backus-Naur Form (ABNF) grammar defined in RFC 5234 [9]:

`<imsc1_key> = ("region_" / "content_") + imsc1_attribute`

`imsc1_attribute` – This part shall be the name of any IMSC1-defined attribute.

`<imsc1_value>` – The value data type range of values and their encodings shall be those supported by IMSC1 for the attribute named in `imsc1_attribute`.

The second part of `<imsc1_key>` shall indicate "region_" when the `<imsc1_value>` applies to regions, and "content_" when it applies to content (text).

Valid examples of `<imsc1_key>` are "region_backgroundColor" and "content_backgroundColor" indicating the background color of either a region or of content (text), respectively.

9.3.6.2 Caption Display Preference Notification Example

For example, the Receiver notifies the Broadcaster Application that the user has changed their caption display preferences to red text on gray background. All the available 708 parameters along with two IMSC1 parameters are included in this example:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "captionDisplayPrefs",
    "cta708": {
      "characterColor": "#FF0000",
      "characterOpacity": 0.5,
      "characterSize": 100,
      "fontStyle": "MonospacedSerifs",
      "backgroundColor": "#808080",
      "backgroundOpacity": 0.25,
      "characterEdge": "Raised",
      "characterEdgeColor": "#000000",
      "windowColor": "#000000",
      "windowOpacity": 0
    },
    "imsc1": {
      "region_textAlign": "center",
      "content_fontWeight": "bold"
    }
  }
}
```

9.3.7 Audio Accessibility Preference Change Notification API

The Audio Accessibility Preference Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the user changes accessibility settings for either video description service and/or audio/aural representation of emergency information (EI).

The Accessibility Preference Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "audioAccessibilityPref" as described below.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["audioAccessibilityPref"]},
    "videoDescriptionService": {
      "type": "object",
      "properties": {
        "enabled": {"type": "boolean"},
        "language": {"type": "string"}
      },
      "required": ["enabled"]
    },
    "audioEIService": {
      "type": "object",
      "properties": {
        "enabled": {"type": "boolean"},
        "language": {"type": "string"}
      },
      "required": ["enabled"]
    }
  },
  "required": ["msgType"], "minProperties": 2
}
```

`videoDescriptionService.enabled` – A Boolean value representing the new state of video description service (VDS) rendering.

`videoDescriptionService.language` – A string indicating the preferred language of VDS rendering, coded according to RFC 5646 [17]. This property shall be present in the notification when `videoDescriptionService.enabled` is equal to `true` and the preferred language of VDS rendering is available at the Receiver.

`audioEIService.enabled` – A Boolean value representing the new state of audio/aural representation of emergency information rendering.

`audioEIService.language` – A string indicating the preferred language of audio/aural representation of emergency information rendering, coded according to RFC 5646 [17]. This property shall be present in the notification when `audioEIService.enabled` is equal to `true` and the preferred language of audio/aural representation of emergency information rendering is available at the Receiver.

For example, if the user has changed the video description service's accessibility preference to ON, the Receiver notifies the Broadcaster Application the current state of video description service and the VDS language preference (when present) as shown below:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "audioAccessibilityPref",
    "videoDescriptionService": {
      "enabled": true,
      "language": "en"
    }
  }
}
```

9.3.8 Alerting Change Notification API

The Alerting Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if there is a change to the version of the AEAT or OSN alerting data structure and the Broadcaster Application has subscribed to receive such notifications via the API specified in Section 9.7.5. Note that the receipt of a new alerting object without a previous receipt is considered a version change. The Alerting Change Notification may also be issued if alerting event filtering has been changed resulting in a change to the filtered events.

The notification message contains a list of URLs referencing the new or updated alerting fragments.

The Alerting Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` key with a value "alertingChange" and a key named `alertList` which is an array of items containing an `alertingType` as defined in the enumerated list, the metadata fragment in XML format and a `receiveTime` if the `alertingType` signals an OSN fragment.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["alertingChange"]},
    "alertList": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "alertingType": {"type": "string", "enum": ["AEAT", "OSN"]},
          "alertingFragment": {"type": "string", "format": "xml"},
          "receiveTime": {"type": "string", "format": "xs:dateTime"},
          "filteredEventList": {
            "type": "array",
            "items": {
              "aeald": {"type": "string"}
            }
          }
        }
      }
    },
    "required": ["alertingType", "alertingFragment"],
    "allOf": [
      {
        "if": {
          "properties": {
            "alertingType": {"enum": ["OSN"]}
          }
        },
        "then": {"required": ["receiveTime"]}
      }
    ]
  },
  "required": ["msgType", "alertList"]
}

```

alertingType – A required parameter containing one of "AEAT" or "OSN". The corresponding **alertingFragment** shall contain the XML fragment corresponding to the **alertingType**.

alertingFragment – A required string shall contain the alerting XML fragment for the associated **alertingType**. The AEAT XML and OSN XML fragments are extracted from their respective LLS tables which are described in A/331 [1].

receiveTime – The date and time when the OSN fragment was received. This value shall be provided when the **alertingType** is "OSN" and is optional otherwise. (Note: The OSN table includes a Notification Duration field which is the duration of the **KeepScreenClear** message starting from the time the OSN was received. Thus, the time the OSN was received is necessary for the Broadcaster Application to fully utilize the OSN information.) The **xs:dateTime** XML data type shall be formatted as defined in the W3C XML Schema [28].

filteredEventList – Provides a list of AEA events that have been filtered by the Receiver. The Receiver may elect to filter an event for a variety of reasons based on user preferences, location or some other criteria. If an AEA event is filtered, the corresponding **AEAT.AEA@aeaId** shall appear in the **filteredEventList** property. If an AEA event is not filtered, the corresponding

AEAT.AEA@aeaId shall not appear in the list. An empty or absent filteredEventList indicates that no events have been filtered by the Receiver. This property is only applicable when the alertingType is "AEAT".

Note that if the filtering criteria change due to, for example, the Receiver moving, or the user changing preferences, previously filtered events may become unfiltered and events that were previously unfiltered may now be filtered. In this case, the Alerting Change Notification shall be issued by the Receiver with a new filteredEventList. The Broadcaster Application may need to take alternative actions based on the new list of events.

aeaId – The AEAT.AEA@aeaId of an AEA event that has been filtered by the Receiver. See the previous description of the filteredEventList property for the usage of this property.

For example, the Receiver may indicate that a new AEAT has been received by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "alertingChange",
    "alertList": [
      { "alertingType": "AEAT",
        "alertingFragment": "<AEAT>...</AEAT>" }
    ]
  }
}
```

As a further example, the Receiver may indicate that a new AEAT and OSN have been received by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "alertingChange",
    "alertList": [
      { "alertingType": "AEAT",
        "alertingFragment": "<AEAT>...</AEAT>" },
      { "alertingType": "OSN",
        "alertingFragment": "<OSN>...</OSN>",
        "receiveTime": "2017-01-01T23:54:59.590Z" }
    ]
  }
}
```

9.3.9 Content Change Notification API

The Content Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if a new signed package or new version of a signed package has been received and the Broadcaster Application has subscribed to receive such notifications via the API specified in Section 9.4.10. Note that a signed package is considered “received” if the contained files are available through the Receiver Web Server.

The notification message contains a list of URLs referencing the received packages.

The Content Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` key with a value "contentChange" and a key named `packageList` which is an array of URLs obtained from the `EFDt.FDT-Instance.File@Content-Location` attribute associated with each package.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["contentChange"]},
    "packageList": {
      "type": "array",
      "items": {"type": "string", "format": "uri"}
    }
  },
  "required": ["msgType", "packageList"]
}
```

`packageList` – An array of package URLs whose contents have been received, checked for signing and are now available for Broadcaster Application access. The package URL of a specific package delivered over ROUTE is signaled in the `EFDt.FDT-Instance.File@Content-Location` attribute according to A/331 [1]. Broadcaster Applications may use these package URLs to determine which collection of files have been received or updated.

For example, to notify the Broadcaster Application that new versions of various content files from a particular package have been received, the content signing has been verified and the files are now available through the Receiver Web Server, the Receiver may issue the following JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "contentChange",
    "packageList": ["items/zz/content"]
  }
}
```

The Broadcaster Application may elect to reload itself or any portion of itself when such a notification is received.

9.3.10 Service Guide Change Notification API

The Service Guide Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if there is a change to some portion of the service guide data structures and the Broadcaster Application has subscribed to receive such notifications via the API specified in Section 9.7.5. Note that the receipt of a new service guide fragment without a previous receipt is considered a version change.

The notification message contains a list of URLs referencing the new or updated service guide XML fragments.

The Service Guide Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` key with a value "seviceGuideChange" and a key named `urlList` which is an array of items containing an `sgType` as defined in the enumerated list and a URL to the associated metadata fragment.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "string",
      "enum": ["serviceGuideChange"]
    },
    "urlList": {
      "type": "array",
      "items": { "type": "object",
        "properties": {
          "sgType": {"type": "string", "enum": ["Service", "Schedule", "Content"]},
          "sgUrl": {"type": "string", "format": "url"}
        },
        "required": ["sgType", "sgUrl"]
      }
    },
    "required": ["msgType", "urlList"]
  }
}
```

`sgType` – One of "Service", "Schedule" or "Content" is required. The corresponding `sgUrl` can be used to access the XML fragment corresponding to the `sgType`. Note that the `urlList` may contain multiple entries for each of these fragments corresponding to the various services, schedules and content descriptions. The Broadcaster Application is expected to process the XML fragments and make the appropriate associations based on the various ID references within the fragments.

`sgUrl` – A required fully-qualified URL that can be used by the Broadcaster Application, for example in an XHR request, to retrieve the current service guide XML fragment for the associated `sgType`.

For example, the Receiver may indicate that a new Schedule has been received by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "serviceGuideChange",
    "urlList": [
      { "sgType": "Schedule",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Schedule.xml" }
    ]
  }
}
```

Note that the URLs provided are examples only. The actual URLs used, including the file names, are completely dependent on the Receiver implementation and how it chooses to make the ESG files available through its HTTP server.

As a further example, the Receiver may indicate that a new Service information and associated Schedule and Content have been received by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "serviceGuideChange",
    "urlList": [
      { "sgType": "Service",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Service.xml" },
      { "sgType": "Schedule",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Schedule.xml" },
      { "sgType": "Schedule",
        "sgUrl": "http://127.0.0.1:8080/wmbc.appctx/Content.xml" }
    ]
  }
}
```

The prefix shown in these examples are informative only. The Broadcaster Application should make no assumptions regarding the path and simply use it to access the fragment data directly.

The referenced service guide files, in this example, *Service.xml*, *Schedule.xml* and *Content.xml*, shall contain the Service, Schedule and Content XML fragments as described in A/332 [2], respectively. The Receiver shall extract each XML fragment from the binary SGDU structure before making it available to the Broadcaster Application.

To associate ESG files with Broadcaster Applications, the corresponding Application Context Identifiers shall be provided in the Extended FDT (EFDT) element, **FDT-Instance@appContextIdList** defined when sending the ESG files in the LCT channel of the ESG Service ROUTE session. Descriptions of the FDT extensions and the ESG Service can be found in A/331 [1]. Application Context Identifiers need not be included in the EFDT if the ESG data is not needed by the Broadcaster Application.

9.3.11 Signaling Data Change Notification API

The Signaling Data Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if a new version of any LLS table or SLS fragment is received since either the Broadcaster Application subscribed to receive such notifications via the API

specified in Section 9.7.5 or a version was previously notified. The Broadcaster Application may respond to the notification of a change to the signaling data by using the Query Signaling Data API specified in Section 9.2.10 to fetch a new copy.

The Signaling Data Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "signalingData"

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["signalingData"]},
  },
  "required": ["msgType"]
}
```

The following is an example notification:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "signalingData",
  }
}
```

9.4 Cache Request APIs

The Cache Request APIs may be used by the currently executing Broadcaster Application to request that the Receiver download one or more objects from a broadband server and place them into a specified location in the Application Context Cache. Files may be identified individually by URL, or a DASH MPD or Period may be specified, in which case all the media files referenced by the MPD or Period are requested.

9.4.1 Cache Request API

The Broadcaster Application can use the Cache Request API to request that the Receiver download one or more indicated files. The Broadcaster Application might request to download ad content via broadband before the time of an ad replacement to avoid playback problems that might occur due to network congestion if the ad were to be streamed in real time. Note that retrieval and storage of broadcast delivered NRT personalized content may be managed by the Filter Codes APIs (see Section 9.11). The Cache Request API includes Filter Codes as well to aid in the Receiver's management of objects in the Application Context Cache.

The Receiver's response to the Cache Request API indicates whether or not the indicated files are already present in the Application Context Cache. Thus, the API may also be used to check whether or not the one or more indicated files are present in the Application Context Cache. The status check function works for files that might have arrived by either the broadcast or the broadband delivery path.

As stated in Section 6.2, storage capability and management of the Application Context Cache are receiver-specific, so that files requested via this API might or might not be stored, depending on the status of the Application Context Cache. However, the Broadcaster Application can use the

Query Cache Usage API defined in Section 9.5 to check how much storage quota of Application Context Cache is assigned for the Application Context ID. The Mark Unused API defined in Section 9.9 can be used to indicate to the Application Context Cache system that cached file(s) are unused. If the currently executing Broadcaster Application is terminated, the Receiver may cancel all in-progress file retrieval processes and release all cached files requested by this API.

Note that the method name, "org.atsc.CacheRequest", starts with a capital "C" inconsistent with the method naming in other parts of this standard. The reader is cautioned to use the method name verbatim to avoid issues.

The Cache Request API shall be defined as follows;

method: "org.atsc.CacheRequest"

params: A JSON object containing the parameters of the method;

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "sourceURL": {"type": "string", "format": "uri"},
    "targetURL": {"type": "string", "format": "uri"},
    "URLs": {"type": "array", "items": {"type": "string", "format": "uri"}},
    "filters": {
      "type": "array",
      "items": {"type": "integer"}
    }
  },
  "required": ["URLs"]
}
```

sourceURL – When **sourceURL** is present, this API requests the Receiver to retrieve the files referenced in the provided **URLs** array, where **sourceURL** is the base URL of each file. When **sourceURL** is present, it shall include the https protocol identifier. When **sourceURL** is absent, the API shall indicate a request for the Receiver to return information about the presence or absence of the identified files within the Application Context Cache.

targetURL – This relative URL shall indicate the location within the Application Context Cache relative to its base where the files are to be placed. When **sourceURL** is not present, the **targetURL** shall indicate the location within the Application Context Cache relative to its base where the Receiver should look for the files given in the **URLs** array and reply with an indication of whether or not all the files are present. When **sourceURL** is present and **targetURL** is not present, the files shall be stored under each URL relative to the root of the Application Context Cache.

URLs – When **sourceURL** is present, the **URLs** array shall represent an array of one or more strings which contain relative URLs of files to be retrieved and stored in the Application Context Cache. Each URL string in **URLs** shall be a relative URL. The effective URL for retrieval of the file from the broadband server shall be a concatenation of the **sourceURL** and the URL string of the file. When **sourceURL** is not present, each URL string shall refer to a file that may be present in the Application Context Cache as the concatenation of the **targetURL** and the

URL of the file, and the response to the API shall indicate whether or not all referenced files are present in the cache.

filters – An array of one or more unsigned integers associated with personalization categories as determined by the broadcaster. It is the broadcaster’s responsibility to maintain a scope of uniqueness of Filter Codes to be within an `AppContextID`.

The response of the Cache Request API shall be defined as follows;

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "cached": {"type": "boolean"}
  },
  "required": ["cached"]
}
```

cached – This required Boolean result shall indicate, when “true” that all the files referenced in the URLs are present in the Application Context Cache and that none are expired. When “false,” **cached** shall indicate that one or more files are not present or are expired.

error: The following error codes may be returned:

- -15: The URL format specified in `sourceURL` or `targetURL` of the request is illegal.
- -16: The URL format specified in one or more URLs of the request is illegal.

For example, the Broadcaster Application may wish to request the download of three PNG files from a broadband server at `http://foo.com/service1` and to store these files in the Application Context Cache in specified subdirectories at or below an `images/` subdirectory. The Filter Codes to be associated with the files are 1007 and 1009. The source and destinations for each of these three files are as follows:

1. File 1:
 - Source: `http://foo.com/service1/A/big-image1.png`
 - Target location in App Context Cache: `images/A/big-image1.png`
2. File 2:
 - Source: `http://foo.com/service1/B/big-image2.png`
 - Target location in App Context Cache: `images/B/big-image2.png`
3. File 3:
 - Source: `http://foo.com/service1/C/big-image3.png`
 - Target location in App Context Cache: `images/C/big-image3.png`

To accomplish the download, the Broadcaster Application could issue the following API:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.CacheRequest",
  "params": {
    "sourceURL": "https://foo.com/service1/",
    "targetURL": "images/",
    "URLs": ["A/big-image1.png", "B/big-image2.png", "C/big-image3.png"],
    "filters": [1007, 1009]
  },
  "id": 37
}
```

In this example, the first PNG file is fetched using the URL `https://foo.com/service1/A/big-image1.png` and placed into the Application Context Cache at a subdirectory `images/A/big-image1.png`.

Upon successfully beginning the retrieval process, if the files had not been retrieved previously, the Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"cached": false},
  "id": 37
}
```

Note that `cached` is “false”, indicating that these files are not already present. If all the files had already been present in the Application Context Cache and none had expired, `cached` would have returned “true”, otherwise the Receiver would begin to re-download the files. If the Broadcaster Application wishes later to check to see whether or not the first two of these files have been successfully downloaded, it could issue the following API to the Receiver:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.CacheRequest",
  "params" {
    "targetURL": "images/",
    "URLs": ["A/big-image1.png", "B/big-image2.png"]
  },
  "id": 38
}
```

If both of the indicated files are present and not expired, the Receiver may respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"cached": true},
  "id": 38
}
```

9.4.2 Cache Request DASH API

The Cache Request DASH API may be used by the currently executing Broadcaster Application to indicate to the Receiver that certain files should be retrieved via broadband and stored in the

Application Context Cache. Instead of listing each URL individually, using this API, files are specified either in an MPEG DASH Period XML fragment or in a complete DASH MPD. If a complete DASH MPD is specified, the MPD file and the MPEG DASH segments specified in the MPD file shall be retrieved via broadband and stored. The URL of each MPEG DASH segment file shall be generated according to the MPEG DASH specification [23]. In response to the XLink Resolution API, the Broadcaster Application can provide the same DASH Period XML fragment.

The Cache Request DASH API may also be used to check whether or not the files indicated in the DASH Period or MPD are present in the Application Context Cache and not expired. The status check function works for files that might have arrived by either the broadcast or the broadband delivery path.

Note that the method name, "org.atsc.CacheRequestDASH", starts with a capital "C" inconsistent with the method naming in other parts of this standard. The reader is cautioned to use the method name verbatim to avoid issues.

The Cache Request DASH API shall be defined as follows;

method: "org.atsc.CacheRequestDASH"

params: A JSON object containing the parameters of the method;

params JSON Schema:

```
{
  "type": "object",
  "oneOf": [
    {
      "properties": {
        "sourceURL": {"type": "string", "format": "uri"},
        "targetURL": {"type": "string", "format": "uri"},
        "Period": {"type": "string", "format": "xml"},
        "filters": {"type": "array", "items": {"type": "integer"}}
      },
      "required": ["targetURL", "Period"]
    },
    {
      "properties": {
        "sourceURL": {"type": "string", "format": "uri"},
        "targetURL": {"type": "string", "format": "uri"},
        "mpdFileName": {"type": "string"},
        "filters": {"type": "array", "items": {"type": "integer"}}
      },
      "required": ["mpdFileName"]
    }
  ]
}
```

When the API is used with a DASH Period:

sourceURL: When sourceURL is present, this API requests the Receiver to retrieve the media files referenced in the provided DASH Period, where sourceURL is the base URL of the files specified in the URLs in the Period. When sourceURL is present, it shall include the https protocol identifier. When sourceURL is absent, the API shall

indicate a request for the Receiver to return information about the presence or absence of the identified files within the Application Context Cache.

targetURL: This relative URL shall indicate the location within the Application Context Cache relative to its base where the files are to be placed. When **sourceURL** is not present, the **targetURL** shall indicate the location within the Application Context Cache relative to its base where the Receiver should look for the files referenced by the **Period** and reply with an indication of whether or not all the files are present and not expired. When **sourceURL** is present and **targetURL** is not present, the files shall be stored under each URL relative to the root of the Application Context Cache.

Period: The **Period** shall represent an XML segment defined as a Period of MPEG DASH compliant with A/331 [1]. Each Media Segment and Initialization Segment URL is constructed using the processing rules of MPEG DASH [23] subclause 5.6. The **Period** shall use only relative URL references. The **Period@duration** attribute shall be present. When **sourceURL** is included, the URLs in the **Period** shall resolve to media files present on the referenced broadband server.

filters: An array of one or more unsigned integers associated with personalization categories as determined by the broadcaster. It is the broadcaster's responsibility to maintain a scope of uniqueness of Filter Codes to be within an **AppContextID**.

When the API is used with a DASH MPD:

sourceURL: This API requests the Receiver to retrieve the media files referenced in the DASH MPD identified by **mpdFileName**, where **sourceURL** is the URL of the broadband server from which the MPD may be retrieved. When **sourceURL** is present, it shall include the https protocol identifier. When **sourceURL** is absent, the API shall indicate a request for the Receiver to return information about the availability of the files identified by the referenced MPD within the Application Context Cache. When **sourceURL** is absent, the response to the request shall indicate "cached":false if the MPD itself is not present in the Application Context Cache, or if any of the files it references are not present or are expired.

targetURL: When **sourceURL** is present, the API requests the Receiver to retrieve and place the files associated with the indicated MPD, and the MPD itself, into the Application Context Cache. In that case, the **targetURL** shall indicate the location within the Application Context Cache relative to its base where the files are to be placed. The Receiver shall also retrieve the MPD and place it at the location in the Application Context Cache given by **targetURL**. When **sourceURL** is not present, the **targetURL** shall indicate the location within the Application Context Cache relative to its base where the Receiver should look for the MPD and the files referenced by the MPD and reply with an indication of whether or not the MPD and all the files it references are present and not expired. When **sourceURL** is present and **targetURL** is not present, the files shall be stored under each URL relative to the root of the Application Context Cache.

mpdFileName: The required **mpdFileName** shall represent the filename of an MPEG DASH MPD that is compliant with A/331[1]. The URL of each Media Segment and Initialization Segment referenced in the indicated MPD is constructed using the processing rules of MPEG DASH [23] subclause 5.6. The referenced MPD shall include only relative URLs. When **sourceURL** is included, the URLs in the MPD shall

resolve to media files present on the referenced broadband server, and the MPD itself shall be present at the server location indicated in `sourceURL` with the filename given in `mpdFileName`.

filters: An array of one or more unsigned integers associated with personalization categories as determined by the broadcaster. It is the broadcaster's responsibility to maintain a scope of uniqueness of Filter Codes to be within an `AppContextID`.

According to MPEG DASH [23] subclause 5.6.4, "URLs at each level of the MPD are resolved according to RFC 3986 with respect to the **BaseURL** element specified at that level of the document or the level above in the case of resolving base URLs themselves (the document 'base URI' as defined in RFC 3986 [21] Section 5.1 is considered to be the level above the MPD level)." For this API, the `sourceURL` is the document "base URI" on the broadband server, and the `targetURL` is the document "base URI" in the Application Context Cache.

The response of the Cache Request DASH API shall be defined as follows;

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "cached": {"type": "boolean"},
  },
  "required": ["cached"]
}
```

cached – This Boolean result shall indicate, when "true", that all the files referenced in the Period or MPD are present in the Application Context Cache at the indicated location and that none are expired. When "false", **cached** shall indicate that one or more files are expired or not present. When `sourceURL` is present in the request, a result of "true" shall be returned in the case that the indicated files are already present in the Application Context Cache and none are expired.

error: The following error codes may be returned:

- -15: The URL format specified in `sourceURL` or `targetURL` of the request is illegal.
- -17: The format of the MPEG DASH fragment specified in the Period is illegal.
- -18: The referenced MPD file cannot be found.

For example, if the Broadcaster Application wishes to request from a broadband server the fetching of MPEG DASH media segment files corresponding to one Period from a broadband server at `https://wxyz.com/svc4.4/content/`, associate them with Filter Codes 1007 and 1009, and place them into the Application Context Cache at `advertising1/` it could issue the following API:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.CacheRequestDASH",
  "params": {
    "sourceURL": "https://wxyz.com/svc4.4/content/",
    "targetURL": "advertising1/",
    "Period": "<Period start='PT9H' duration='PT30S'>
      <AdaptationSet mimeType='video/mp4' />
      <SegmentTemplate timescale='9000' media='video/xbc$Number$.mp4v'
        duration='90000' startNumber='32401' /><Representation id='v2'
        width='1920' height='1080' /></Period>",
    "filters": [1007, 1009]
  },
  "id": 38
}
```

The resulting video Media Segment files would be retrieved and stored in the Application Context Cache in the `advertising1/video/` subdirectory. Upon successfully beginning the retrieval process, if the files had not been retrieved previously the Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"cached": false},
  "id": 38
}
```

The `cached` value of “false” in the response indicates that the requested files are not all already present in the cache. If all the files had already been present in the Application Context Cache and none were expired, `cached` would have returned “true”, otherwise the Receiver would begin to re-download the files.

If the Broadcaster Application wishes later to check to see whether or not the files associated with the indicated DASH Period have been successfully downloaded, it could issue the following API to the Receiver:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.CacheRequestDASH",
  "params": {
    "targetURL": "advertising1/",
    "Period": "<Period start='PT9H' duration='PT30S'>
      <AdaptationSet mimeType='video/mp4' />
      <SegmentTemplate timescale='9000' media='video/xbc$Number$.mp4v'
        duration='90000' startNumber='32401' />
      <Representation id='v2' width='1920' height='1080' /></Period>"
  },
  "id": 37
}
```

If all of the indicated Media Segment files are present, including Initialization Segments, the Receiver may respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"cached": true},
  "id": 37
}
```

If any of the indicated Media Segment files or Initialization Segments are missing, the Receiver may respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"cached": false},
  "id": 37
}
```

9.5 Query Cache Usage API

If the Broadcaster Application wishes to know the total quota size of the cache assigned to the Application Context ID with which it is associated and the total current usage of the cache in the Application Context ID hierarchy, the Query Cache Usage API can be used.

The Query Cache Usage API shall be defined as follows:

method: "org.atsc.query.cacheUsage"
 params: none

The Receiver's response to the Query Cache Usage API shall be defined as follows;

result: a JSON object containing the result of the method.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "usageSize": {"type": "integer"},
    "quotaSize": {"type": "integer"}
  },
  "required": ["usageSize", "quotaSize"]
}
```

usageSize – The total usage byte size of the cache associated with the Application Context ID of the broadcaster application.

quotaSize – The total size in bytes of the quota allocated for the Application Context ID of the Broadcaster Application.

For example, if the Broadcaster Application wishes to query the status of cache usage, it could do as follows:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.cacheUsage",
  "id": 39
}
```

If the usage size of cache equals 8,475,337 bytes and the quota size available to the Application Context ID equals 209,715,200 bytes, the Receiver might respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "usageSize": 8475337,
    "quotaSize": 209715200
  },
  "id": 39
}
```

9.6 Event Stream APIs

Events intended for Broadcast Applications can be encountered in broadcast media, either as Event Message ('emsg' or 'evti') Boxes in-band with the media, or as static EventStream elements at the period level in a DASH MPD. These Events may initiate interactive actions on the part of a Broadcast Application, or they may indicate that new versions of files are being delivered, or various other things.

In the case of AMP media playback, parsing and processing of Events is expected to be performed by the Broadcaster Application. In the case of RMP media playback, three APIs are needed to support this function:

- Subscribe to an Event Stream
- Unsubscribe from an Event Stream
- Receive an Event from a subscribed Event Stream

9.6.1 Event Stream Subscribe API

A Broadcaster Application that is currently subscribed to Event Stream notifications shall be notified when certain Event Stream events are encountered during RMP playback in the MPD or the Media Segments. For MPEG DASH, the Event Message Box ('emsg') box contains in-band events, and the MPD may include static events in an EventStream element at the period level. Events in MMT-based Services may be carried in 'evti' boxes in MPUs. A Broadcaster Application that wishes to be notified when a particular type of event occurs may register for that type of event using a `schemeIdUri` and optionally an accompanying `value` parameter.

The Event Stream Subscribe API (sent from the Broadcaster Application to the Receiver) shall be defined as follows:

method: "org.atsc.eventStream.subscribe"

params: A JSON object containing a `schemeIdUri` and optionally an accompanying `@value`.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "schemeIdUri": {"type": "string", "format": "uri"},
    "value": {"type": "string"}
  },
  "required": ["schemeIdUri"]
}

```

schemeIdUri – The **schemeIdUri** URI string associated with the Event Stream event of interest to the Broadcaster Application.

value – An optional string used to identify a particular Event Stream event.

For example, if the Broadcaster Application wishes to register for Event Stream events associated with **schemeIdUri** "urn:uuid:9a04f079-9840-4286", it could subscribe as follows:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.eventStream.subscribe",
  "params": {"schemeIdUri": "urn:uuid:9a04f079-9840-4286"},
  "id": 22
}

```

The Receiver might respond with:

```

<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 22
}

```

The Receiver would then be set to communicate any Event Stream events tagged with **schemeIdUri** "urn:uuid:9a04f079-9840-4286" to the Broadcaster Application using the Event Stream Event API defined in Section 9.6.3 below.

If the Broadcaster Application were only interested in Event Stream events associated with this **schemeIdUri** when the accompanying **value** = "17", it could subscribe while including the **value** parameter:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.eventStream.subscribe",
  "params": {
    "schemeIdUri": "urn:uuid:9a04f079-9840-4286",
    "value": "17"
  },
  "id": 23
}

```

The Receiver might respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 23
}
```

The Receiver would then be set to communicate any Event Stream event tagged with `schemaIdUri` "urn:uuid:9a04f079-9840-4286" and `value` = "17" to the Broadcaster Application using the notification API defined in Section 9.6.3 below. The Broadcaster Application would not be notified of Event Stream events tagged with unsubscribed values of `schemaIdUri` or those with a subscribed `schemaIdUri` but not matching any specified `value`.

The Broadcaster Application may subscribe to multiple different Event Stream events (with different `schemaIdUri` values, or different `schemaIdUri/value` combinations).

Once subscribed, the Broadcaster Application may unsubscribe using the API described in Section 9.6.2.

9.6.2 Event Stream Unsubscribe API

If a Broadcaster Application has subscribed to an Event Stream using the Event Stream Subscribe API defined in Section 9.6.1, it can use the Event Stream Unsubscribe API defined here to request that the Receiver discontinue notifications pertaining to the identified event.

The Event Stream Unsubscribe API (sent from the Broadcaster Application to the Receiver) shall be defined as follows:

`method`: "org.atsc.eventStream.unsubscribe"

`params`: A JSON object containing a `schemaIdUri` and optionally an accompanying `@value`.

`params JSON Schema`:

```
{
  "type": "object",
  "properties": {
    "schemaIdUri": {"type": "string", "format": "uri"},
    "value": {"type": "string"}
  },
  "required": ["schemaIdUri"]
}
```

`schemaIdUri` – The `schemaIdUri` URI string associated with the Event Stream event for which the Broadcaster Application would like to remove the subscription.

`value` – An optional string used to identify a particular Event Stream event.

For example, if the Broadcaster Application wishes to unsubscribe to all Event Stream events associated with `schemaIdUri` "urn:uuid:9a04f079-9840-4286", regardless of the value of the `value` parameter, it could use the following API:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.eventStream.unsubscribe",
  "params": {"schemeIdUri": "urn:uuid:9a04f079-9840-4286"},
  "id": 26
}
```

If the operation was successful, the Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 26
}
```

If the Broadcaster Application had subscribed to this same `schemeIdUri` using `value="47"` and `value="48"`, and now wished to unsubscribe to the latter, it could use the following API:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.eventStream.unsubscribe",
  "params": {
    "schemeIdUri": "urn:uuid:9a04f079-9840-4286",
    "value": "48"
  },
  "id": 29
}
```

If the operation were successful, the Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 29
}
```

9.6.3 Event Stream Event API

The Event Stream Event API shall be issued by the Receiver to the currently executing Broadcaster Application during RMP playback if an event is encountered in the content of the currently selected Service or currently playing content that matches the value of `schemeIdUri` (and accompanying value, if it was provided in the subscription) provided in a prior Event Stream Subscription.

The Event Stream Event API shall be as defined below:

method: "org.atsc.eventStream.event"

params: A JSON object conforming to the JSON Schema defined below.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "schemeIdUri": {
      "type": "string",
      "format": "uri"
    },
    "value": { "type": "string" },
    "eventTime": {
      "type": "number",
      "minimum": 0
    },
    "duration": {
      "type": "number",
      "minimum": 0
    },
    "id": {
      "type": "integer",
      "minimum": 0,
      "maximum": 4294967295
    },
    "data": { "oneOf": [
      { "type": "string" },
      { "type": "object" }
    ] }
  },
  "required": ["schemeIdUri", "eventTime"]
}

```

`schemeIdUri` – A required string identifying the Event Stream with which the Event is associated.

`value` – An optional string with semantic as defined by the owners of the Event Stream scheme identified by `schemeIdUri`.

`eventTime` – A required floating-point number representing the presentation time at which the event starts on the RMP media presentation timeline, expressed as an offset in seconds from the `startDate` (as specified in Query RMP Media Time API, section 9.14.1).

`duration` – An optional floating-point number representing the duration of the event in seconds.

`id` – An optional number indicating the relative id of this event.

`data` – An optional string or object representing additional data associated with this event, with semantics as defined by the owners of the Event Stream scheme identified by `schemeIdUri`.

The start time of the event on the RMP presentation timeline is when `currentTime` (as provided in Query RMP Media Time API, Section 9.14.1) is equal to `eventTime` of the Event.

An example Event Stream notification message that might occur if the Broadcaster Application had registered for Event Stream events using a `schemeIdUri` of `tag:xyz.org:evt:xyz.aaa.9`:


```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.eventStream.event",
  "params": {
    "schemeIdUri": "tag:xyz.org:evt:xyz.aaa.9",
    "value": "ev47",
    "eventTime": 1450.6,
    "id": 60,
    "data": "d8a0c98fs08-d9df0809s"
  }
}
```

Note in this example that if the Broadcaster Application had included a value parameter in the subscription, and that parameter had not been "ev47", this particular event would not be forwarded to the Broadcaster Application.

9.7 Request Receiver Actions

9.7.1 Acquire Service API

The current service may be changed by two entities, the Broadcaster Application via request to the Receiver, or the user via the Receiver directly. Depending on the information sent in the Broadcaster Application signaling, the Receiver does one of the following when a new service is successfully selected:

- If the Broadcaster Application signaling indicates that the same Broadcaster Application should be launched for the new service, then the Receiver allows the Broadcaster Application to continue to run and sends the Broadcaster Application a service change notification. (This is also what would happen if the service were changed by the user directly, and the Broadcaster Application signaling indicated that the same Broadcaster Application should be launched for the new service.)
- If the Broadcaster Application signaling indicates that no Broadcaster Application or a different Broadcaster Application should be launched for the new service, then the Receiver will terminate the current Broadcaster Application, if appropriate.

The reason why a Broadcaster Application might request the Receiver to change the service selection might be to jump to another service of the same broadcaster for content that might be of interest to the user. The Receiver processes the request and if it can, it changes the service selection.

The Acquire Service API shall be defined as follows:

method: "org.atsc.acquire.service"

params: the globally unique Service ID of the service to be acquired.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "svcToAcquire": {"type": "string"}
  },
  "required": ["svcToAcquire"]
}
```

`svcToAcquire` – This required string shall correspond to the globally unique Service ID (as defined in **SLT.Service@globalServiceID**; see A/331 [1] Section 6.3) of the service to acquire.

Response:

If the acquisition is successful, the Receiver shall respond with a JSON-RPC response object with a null `result` object. If the acquisition is not successful, the Receiver shall respond with a JSON-RPC response object including one of the following `error` objects (See Table 8.2):

```
"error": {"code": -6, "message": "Service not found"}
"error": {"code": -7, "message": "Service not authorized"}
```

For example, if the Broadcaster Application requests access to a service represented by globally unique Service ID "https://doi.org/10.5239/8A23-2B0B", it can issue this request to the Receiver:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.acquire.service",
  "params": {"svcToAcquire": "https://doi.org/10.5239/8A23-2B0B"},
  "id": 59
}
```

The Receiver would respond, if acquisition were successful with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 59
}
```

If globally unique Service ID "https://doi.org/10.5239/8A23-2B0B" is unknown to the Receiver, the response would be:

```
<-- {
  "jsonrpc": "2.0",
  "error": {"code": -6, "message": "Service not found"},
  "id": 59
}
```

9.7.2 Video Scaling and Positioning API

A Broadcaster Application in an application-enhanced Service (e.g. playing within the video plane that is positioned on top of the video produced by the Receiver Media Player) can use the video scaling and positioning JSON-RPC method to request that the RMP render its video at less than full-scale (full screen), and to position it at a specified location within the display window.

The Video Scaling and Positioning API shall be defined as follows:

`method`: "org.atsc.scale-position"

`params`: the scaling factor and the X/Y coordinates of the upper left corner of the scaled window.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "scaleFactor": {
      "type": "double",
      "minimum": 10.0,
      "maximum": 100.0
    },
    "xPos": {
      "type": "double",
      "minimum": 0.0,
      "maximum": 100.0
    },
    "yPos": {
      "type": "double",
      "minimum": 0.0,
      "maximum": 100.0
    }
  },
  "required": ["scaleFactor", "xPos", "yPos"]
}
```

scaleFactor – This required double value in the range 10.0 to 100.0 shall represent the video scaling parameter, where 100.0 represents full-screen (no scaling);

xPos – This required double value in the range 0.0 to 100.0 shall represent the X-axis location of the left side of the RMP's video window, represented as a percentage of the full width of the screen. A value of 0.0 indicates the left side of the video window is aligned with the left side of the display window. A value of 50.0 indicates the left side of the video window is aligned with the vertical centerline of the display window, etc.

yPos – This required double value in the range 0.0 to 100.0 shall represent the Y-axis location of the top of the RMP's video window, represented as a percentage of the full height of the screen. A value of 0.0 indicates the top of the video window is aligned with the top of the display window. A value of 50.0 indicates the top of the video window is aligned with the horizontal centerline of the display window, etc.

The zero axis of the coordinate system shall be the upper left corner, as with CSS.

The parameter values shall be set such that no portion of the video window would be rendered outside the display window.

For example, if the Broadcaster Application wished to scale the displayed video to 25% of full screen, and position the left edge of the display horizontally at 10% of the screen width and the top edge of the display vertically at 15% of the screen height, it would issue this JSON-RPC API to the Receiver:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.scale-position",
  "params": {
    "scaleFactor": 25.0,
    "xPos": 10.0,
    "yPos": 15.0
  },
  "id": 589
}
```

If scaling/positioning were successful, the Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 589
}
```

If scaling/positioning were not successful, the Receiver would respond with a JSON object including an "error" object:

```
<-- {
  "jsonrpc": "2.0",
  "error": {"code": -8, "message": "Video scaling/position failed"},
  "id": 589
}
```

9.7.3 Set RMP URL API

The Broadcaster Application may choose to use the Receiver Media Player (RMP) to play video content originated from an alternate source (e.g. broadband or locally cached content) instead of the broadcast-delivered content. In this way, the Broadcaster Application can take advantage of an optimized media player provided by the Receiver. The Broadcaster Application may use the Set RMP URL API to request the Receiver to use its RMP to play content originated from a URL provided by the Broadcaster Application. Once the Receiver is notified to play content from the application-provided URL, the RMP stops rendering the broadcast content (or the content being rendered at the time of the request) and begins rendering the content referenced by the new URL.

The content presented via the specified MPD is considered to be a part of the currently selected Service. The effects of changing this URL are temporary and if the Service is re-selected (e.g. by the Broadcaster Application via the Acquire Service API) or if playback of the MPD specified through the Set RMP URL API reaches its end, the RMP shall return to playing the content defined in the service-level signaling. Note that the Broadcaster Application may use the stopRMP operation to stop the current RMP playback, but selecting a service overrides this operation and the RMP shall begin playing the content specified by the service-level signaling of the selected Service.

The Broadcaster Application may request that the RMP synchronize the requested operations to a future time on the current presentation timeline. This explicit synchronization is needed in the scenario where the currently playing content is being delivered by an alternate transport without Xlink support such as what might be found in redistribution. The RMP is not expected to be

capable of queueing more than one such pending request at a time. Synchronization is indicated through use of the `rmplSyncTime` parameter as specified in the following paragraphs.

If the Set RMP URL API is called by a Broadcaster Application and a `startRMP` operation is specified, then:

- if no `rmplSyncTime` value is specified in the current request, then the Receiver is expected to cancel any pending Set RMP URL request and immediately begin playback of the MPD given in the current request;
- if the RMP is currently playing the content specified in service-level signaling and if an `rmplSyncTime` value is specified in the current request, then the Receiver is expected to cancel any pending Set RMP URL request and to begin the playback of the MPD given in the current request when the presentation time specified by `rmplSyncTime` is reached;
- if the RMP is currently playing an MPD specified in a prior Set RMP URL request and if `rmplSyncTime` has the value -1.0 specified in the current request, then the Receiver is expected to cancel any pending Set RMP URL request and to begin the playback of the MPD given in the current request when the end of the presentation currently being played by the RMP is reached;
- Otherwise, the Receiver shall ignore the current request and continue to play the current content.

If the Set RMP URL API is called by a Broadcaster Application and a `stopRMP` operation is specified, then:

- if the RMP is currently playing the content specified in service-level signaling or an MPD specified in a prior Set RMP URL request and if no `rmplSyncTime` value is specified in the current request, then the Receiver is expected to cancel any pending Set RMP URL request and immediately stop the presentation of the RMP;
- if the RMP is currently playing the content specified in service-level signaling or an MPD specified in a prior Set RMP URL request and if an `rmplSyncTime` value is specified in the current request, then the Receiver is expected to cancel any pending Set RMP URL request and to continue playback of the current content until the presentation time indicated by `rmplSyncTime` is reached, at which time it is expected to stop the presentation of the RMP; and
- if the RMP playback is currently stopped, then the Receiver shall ignore the current request.

If the Set RMP URL API is called by a Broadcaster Application and a `resumeService` operation is specified, then:

- if the RMP is currently either playing an MPD specified in a prior Set RMP URL request or stopped by a `stopRmp` operation of a prior Set RMP URL request and if no `rmplSyncTime` is specified in the current request, then the Receiver is expected to cancel any pending Set RMP URL request and immediately play the content specified in service-level signaling;
- if the RMP is currently playing an MPD specified in a prior Set RMP URL request and if an `rmplSyncTime` is specified in the current request, then the RMP is expected to immediately cancel any pending Set RMP URL request and to continue playback of the currently playing MPD until the presentation time indicated by `rmplSyncTime` is reached, at which time it is expected to return to playing the content defined in the service-level signaling;

- Otherwise, the Receiver shall ignore the current request and continue to play the current content.

If the receiver determines that it is unable to perform the action as requested, it shall return an error code and shall not perform the requested action.

At the time the RMP begins playback of the MPD given in a Set RMP URL request, the Broadcaster Application can receive a notification via the Signaling Data Change Notification API (Section 9.3.11). In any case, whenever an MPD change or update causes a discontinuity in the presentation timeline, the RMP shall cancel any pending Set RMP URL requests.

The Broadcaster Application specifies the content to be played by the RMP by providing the URL of an MPD. The MPD shall be constructed in accordance with A/331 [1].

The URL may include an MPD Anchor identifying the entry point on the media presentation timeline (e.g. an offset from the start of the MPD, an offset from the start of a named period, a UTC time, or the “live edge”) at which the RMP should begin playback. MPD Anchor shall be as defined in MPEG DASH [23]. This allows flexibility for many use cases including bookmarking. If the playback position indicated by a specified MPD Anchor is not available to the RMP, the RMP shall not play the MPD at the given URL and an error code shall be returned.

The SET RMP URL API shall be defined as follows:

method: "org.atsc.setRMPURL"

params: A JSON object consisting of keys named `operation`, `rmpurl`, and a `rmpSyncTime` that describe a behavior to be performed by the RMP with respect to content playback.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "operation": {
      "type": "string",
      "enum": ["startRmp", "stopRmp", "resumeService"]
    },
    "rmpurl": { "type": "string", "format": "uri" },
    "rmpSyncTime": { "type": "number" }
  }
}
```

`operation` – This optional string shall define the operation to be performed by the RMP. When `operation` is absent, the Receiver shall cancel any pending Set RMP URL API request. The meaning of the enumerated values shall be defined as follows:

"startRmp" indicates that the RMP shall start playing the URL provided by the `rmpurl` property as described in the property description below.

"stopRmp" indicates that the RMP shall cease playback. For this operation, the "rmpurl" property is not required and shall be ignored if present.

"resumeService" indicates that the RMP shall resume normal playback of the current Service. For this operation, the "rmpurl" property is not required and shall be ignored if present.

`rmpurl` – When the operation value is set to "startRmp", this string shall be specified and provide a URL referencing an MPD to be played by the RMP. The URL shall be accessible to the

Receiver. The appropriate error code (see below) will be returned if the URL cannot be accessed.

`rmpSyncTime` – This optional floating-point number indicates a future time (i.e. later than `currentTime`) on the media presentation timeline of the presentation currently being played by the RMP (in seconds, relative to the media presentation time given by `startDate` as specified in the Query RMP Media Time API, Section 9.14.1) at which the action specified by `operation` should be performed. If `rmpSyncTime` is not specified, the action indicated by `operation` should begin playing immediately. If `rmpSyncTime` has the value -1.0, the action indicated by `operation` should be performed when the end of the presentation currently being played by the RMP is reached. (The end of the presentation is considered to occur when no further presentation description is indicated.)

Response:

result: A null object upon success.

error: The following error codes may be returned:

- -11: The indicated MPD cannot be accessed
- -12: The content cannot be played
- -13: The requested MPD Anchor cannot be reached
- -19: The synchronization specified by `rmpSyncTime` cannot be achieved
- -21: Changing RMP playback from the current source is not supported

For example, if the Broadcaster Application requests the RMP to play content from a broadband source at a DASH server located at `http://stream.wxyz.com/33/program.mpd`, it can issue a command to the Receiver as follows:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setRMPURL",
  "params": {"operation": "startRmp",
             "rmpurl": "http://stream.wxyz.com/33/program.mpd"},
  "id": 104
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 104
}
```

If the Receiver's RMP cannot play the content, the Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "error": { "code": -12, "message": "The content cannot be played"},
  "id": 104
}
```

Furthering the example, the user interacts with a Broadcaster Application, which now wishes to display a full-screen video on demand selection screen with no video. The Broadcaster Application makes the following request to the Receiver:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setRMPURL",
  "params": {"operation": "stopRmp"},
  "id": 113
}
```

Upon success, the RMP would cease displaying video and the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 113
}
```

Finally, the user exits from the video on demand scenario and now wants to return to watching broadcast services. The Broadcaster Application would make the following request:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setRMPURL",
  "params": {"operation": "resumeService"},
  "id": 106
}
```

Upon success, the Receiver would resume normal playback operations and would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 106
}
```

As a fourth example, if the `currentTime` as provided by the Query RMP Media Time API is 1740 seconds past the `startDate`, no MPD given in a prior Set RMP URL request is being played, and the Broadcaster Application wants the RMP to begin the playback of the MPD located at `http://stream.wxyz.com/33/program.mpd` when the `currentTime` of the presentation currently being played by the RMP reaches 1800 seconds with an entry point into the specified MPD that is 5 minutes from its beginning, the application can issue a command to the Receiver as follows:


```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setRMPURL",
  "params": { "operation": "startRmp",
              "rmpurl": "http://stream.wxyz.com/33/program.mpd#t=5:00",
              "rmpSyncTime": 1800.00 }
  "id": 107
}
```

Upon successfully scheduling the pending MPD playback, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 107
}
```

If the Receiver's RMP cannot accept the request for scheduled playback of the content (e.g. because the specified synchronization time has passed or is too soon for the RMP to prepare), the Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "error": { "code": -19,
            "message": "The synchronization specified by rmpSyncTime cannot be
achieved" },
  "id": 107
}
```

As a fifth example, if the RMP is playing an MPD per a prior Set RMP URL API request and the `currentTime` is 10 seconds past the `startDate` as provided by the Query RMP Media Time API, and the Broadcast Application wants the RMP to resume the playback of the content specified in service-level signaling when the `currentTime` of the presentation currently being played reaches 60 seconds, it can issue a request to the Receiver as follows:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setRMPURL",
  "params": { "operation": "resumeService",
              "rmpSyncTime": 60.00 }
  "id": 108
}
```

Upon successfully scheduling the requested playback of the content specified in the service-level signaling, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 108
}
```

As a sixth example, if the RMP is playing content specified in the service-level signaling (or an MPD specified in a prior Set RMP URL API), the `currentTime` is 10 seconds past the `startDate` as provided by the Query RMP Media Time API, then the Broadcaster Application can request the Receiver to stop the current playback when the `currentTime` reaches 30 seconds with the following request:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setRMPURL",
  "params": {"operation": "stopRmp",
             "rmpSyncTime": 30.00}
  "id": 109
}
```

Upon successfully scheduling the requested stop of the current playback, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 109
}
```

9.7.4 Audio Volume API

By default, the audio output of the Receiver Media Player and that of the User Agent are mixed. The Broadcaster Application may set and get the volume of the HTML5 media element using the `.volume` property. It may wish to set and get the audio volume of the Receiver Media Player. For example, the Broadcaster Application might mute the audio output of broadcast service when the user chooses to watch broadband content rendered with an HTML5 media element. The Audio Volume API may be used for such a case.

Figure 9.1 illustrates audio processing in an example receiver in which the audio output of the User Agent is mixed with the audio output of the Receiver Media Player for presentation to the user. The Broadcaster Application controls the volume of its output using the `.volume` property of the `HTMLMediaElement`. Analogously, the Audio Volume API defined here may be used to set the volume of the Receiver Media Player, shown as “V1” in the figure. Note that the API changes *only* the RMP volume (“V1”).

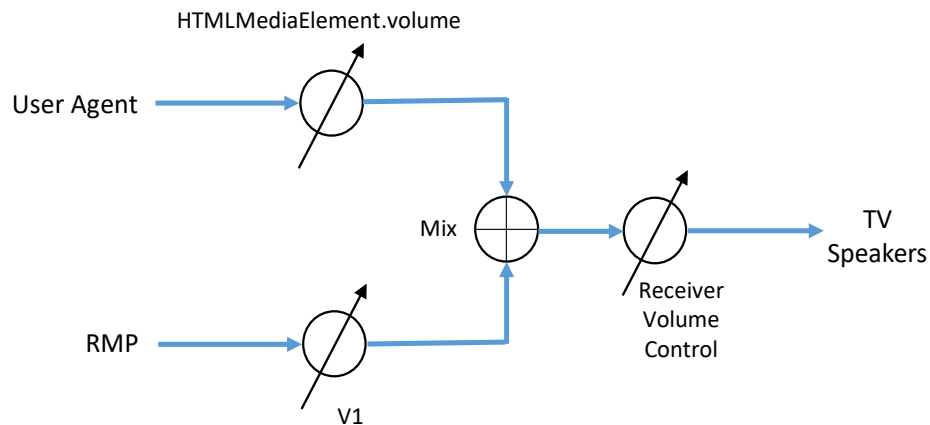


Figure 9.1 RMP audio volume.

If a volume element is provided in the request, the Receiver processes the request to set the RMP volume. The Receiver's response provides the current volume in either case.

The Audio Volume API shall be defined as follows:

method: "org.atsc.audioVolume"

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "audioVolume": {"type": "number"},
  }
}

```

`audioVolume` – This optional floating-point number in the range 0 to 1, when present, shall correspond to a value of audio volume to be set in the Receiver Media Player. The value of number shall be from 0.0 (minimum or muted) to 1.0 (full volume). The encoding is the same as the `.volume` property of the HTML5 media element. If volume is not specified in the request, the volume is not changed by this request. This can be used to determine the current volume setting.

Response:

result: A JSON object containing an `audioVolume` key whose value represents the current audio level from 0.0 (minimum or muted) to 1.0 (full volume).

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "audioVolume": {"type": "number"}
  },
  "required": ["audioVolume"]
}

```

audioVolume – This floating-point number in the range 0 to 1 shall indicate the current audio volume of the Receiver Media Player, where 0 indicates minimum volume or muted, and 1.0 indicates full volume.

For example, if the Broadcaster Application wishes for the Receiver Media Player set the audio volume to half volume (50%):

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.audioVolume ",
  "params": {"audioVolume ": 0.5},
  "id": 239
}
```

If the request is processed successfully, the Receiver might respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"audioVolume ": 0.5},
  "id": 239
}
```

9.7.5 Integrated Subscribe / Unsubscribe API for Notifications

The various types of notifications that the Receiver may provide are described throughout Section 9. The Broadcaster Application may wish to receive specific notifications. When a Broadcaster Application starts, no notifications shall be subscribed, and the Receiver shall not send any notifications. The Broadcaster Application may use the subscription API to begin receiving the desired notifications. The Receiver shall send subscribed notifications until the Broadcaster Application requests the Integrated Unsubscribe API. In addition, once the Broadcaster Application has been terminated, the Receiver shall automatically unsubscribe all the subscribed notifications that were requested by the Broadcaster Application.

Two APIs are needed to support this function:

- Integrated Subscribe API
- Integrated Unsubscribe API

Table 9.3 describes the list of msgTypes to subscribe to each of the notifications. The msgType column list is identical to the enum parameter of the subscribe and unsubscribe APIs.

Note that the event stream notification and its associated subscription methods are specified in Section 9.6. This separate interface allows parameters to be specified to filter various event streams.

Table 9.3 Subscription Parameter List

Notification APIs	Reference	msgType
All Notification APIs	-	All
Rating Change Notification API	9.3.1	ratingChange
Rating Block Change Notification API	9.3.2	ratingBlock
Service Change Notification API	9.3.3	serviceChange
Caption State Change Notification API	9.3.4	captionState
Language Preference Change Notification API	9.3.5	languagePref
Caption Display Preferences Change Notification API	9.3.6	CCDisplayPref

Audio Accessibility Preference Change Notification API	9.3.7	audioAccessPref
Alerting Change Notification API	9.3.8	alertingChange
Content Change Notification API	9.3.9	contentChange
Service Guide Change Notification API	9.3.10	serviceGuideChange
Signaling Data Change Notification API	9.3.11	signalingData
Content Recovery State Change Notification API	9.10.4	contentRecoveryStateChange
Display Override Change Notification API	9.10.5	displayOverrideChange
Recovered Component Info Change Notification API	9.10.6	recoveredComponentInfoChange
RMP Media Time Change Notification API	9.14.5	rmpMediaTimeChange
RMP Playback State Change Notification API	9.14.6	rmpPlaybackStateChange
RMP Playback Rate Change Notification API	9.14.7	rmpPlaybackRateChange
DRM Notification API	9.15.1	DRM
XLink Resolution Notification API	9.16.1	xlinkResolution

9.7.5.1 Integrated Subscribe API

The Integrated Subscribe API for Asynchronous Notification Changes shall be defined as follows:

method: "org.atsc.subscribe"

params: A JSON object consisting of a key named `msgType` with multiple `enum` values representing notifications to which the Broadcaster Application wants to subscribe.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": [List of msgTypes Column in Table 9.3]
      }
    }
  },
  "required": ["msgType"]
}
```

Response:

result: A JSON object containing the `enum` values of `msgType` that successfully subscribed.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": [List of msgTypes Column in Table 9.3]
      }
    }
  },
  "required": ["msgType"]
}

```

For example, the Broadcaster Application wants to subscribe the notification for “alertingChange”, “ratingBlock” and “contentChange”:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.subscribe",
  "params": {
    "msgType": ["alertingChange", "ratingBlock", "contentChange"]
  },
  "id": 51
}

```

Upon success, the Receiver would respond:

```

<-- {
  "jsonrpc": "2.0",
  "result": {
    "msgType": ["alertingChange", "ratingBlock", "contentChange"]
  },
  "id": 51
}

```

If the Broadcaster Application may attempt to subscribe to all notifications:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.subscribe",
  "params": {
    "msgType": ["All"]
  },
  "id": 51
}

```

If the Receiver supports all functionality except content recovery, it may respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "msgType": ["ratingChange", "ratingBlock", "serviceChange",
"captionState", "languagePref", "captionDisplayPrefs",
"audioAccessibilityPref", "altertingChange", "contentChange",
"rmpMediaTimeChange", "rmpPlaybackStateChange", "rmpPlaybackRateChange",
"DRM", "xlinkResolution"]
  },
  "id": 51
}
```

9.7.5.2 Integrated Unsubscribe API

The Integrated Unsubscribe API for Asynchronous Notification Changes shall be defined as follows:

method: "org.atsc.unsubscribe"

params: A JSON object consisting of a key named `msgType` with multiple `enum` values representing notifications from which the Broadcaster Application wants to unsubscribe.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": [List of Keywords Column in Table 9.3]
      }
    }
  },
  "required": ["msgType"]
}
```

Response:

result: A JSON object containing the `enum` values of `msgType` that were successfully unsubscribed.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": [List of Keywords Column in Table 9.3]
      }
    }
  },
  "required": ["msgType"]
}

```

For example, the Broadcaster Application wants to unsubscribe from all notifications:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.unsubscribe",
  "params": {
    "msgType": ["All"]
  },
  "id": 52
}

```

Upon success, the Receiver would respond with the list of msgTypes from which the Broadcaster Application successfully unsubscribed:

```

<-- {
  "jsonrpc": "2.0",
  "result": {
    "msgType": ["alertingChange", "contentChange",
"contentRecoveryStateChange", "displayOverrideChange",
"recoveredComponentInfoChange", "rmpMediaTimeChange",
"rmpPlaybackStateChange", "rmpPlaybackRateChange"]
  },
  "id": 52
}

```

9.8 Media Track Selection API

The Broadcaster Application may request the Receiver's Receiver Media Player to select a particular video stream available in the Service, for example an alternate camera angle. Alternatively, it might request the Receiver Media Player to select an audio presentation other than the one it would have chosen based on the user's preferences. The Media Track Selection API may be used for these cases.

The Receiver processes the request and if it can, it changes the selection.

The Media Track Selection API shall be defined as follows:

method: "org.atsc.track.selection"

params: an id value associated with the DASH Period.AdaptationSet element to be selected, or alternatively, for complex audio presentations involving pre-selection, the

DASH `Period.Preselection@id` value. For unambiguous selection of one track or audio presentation, all `id` values within the `Period` should be unique.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "selectionId": {"type": "integer"}
  },
  "required": ["selectionId"]
}
```

`selectionId` – This required integer shall correspond to a value of `@id` attribute in either an `AdaptationSet` in the current `Period`, or an `@id` attribute in a `Preselection` element in the current `Period`.

For example, if the Broadcaster Application wishes for the Receiver Media Player to find and select a video `AdaptationSet` with an `id` value of 5506, it could send the following WebSocket message:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.track.selection",
  "params": {"selectionId": 5506},
  "id": 29
}
```

If the requested `AdaptationSet` was successfully selected, the Receiver would respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 329
}
```

If the requested track cannot be selected, the Receiver shall respond with error code -10:

```
<-- {
  "jsonrpc": "2.0",
  "error": {"code": -10, "message": "Track cannot be selected"},
  "id": 329
}
```

9.9 Mark Unused API

The Mark Unused API may be used by the currently executing Broadcaster Application to indicate to the Application Context Cache that an element within the cache is unused. The Receiver may then perform the appropriate actions to reclaim the resources used by the unused element.

The Mark Unused API shall be defined as follows:

method: "org.atsc.cache.markUnused"

params: A JSON object consisting of a key named `elementUri` with the URI of the element that should be marked unused in the Broadcaster Application's Application Context Cache as follows:

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "elementUri": {"type": "string", "format": "uri"}
  },
  "required": ["elementUri"]
}
```

For example, the Broadcaster Application may wish to indicate that a particular replacement ad was not needed anymore after it had been used. The files comprising the replacement ad, that is the MPD and the associated audio and video segments, would be sent in a particular directory hierarchy labeled "ads/16" for this example. The Broadcaster Application would mark the MPD file as unused and all segments referenced by the MPD as unused as well. Note that the Broadcaster Application would be responsible for marking all of the resources of the ad as unused, the Receiver is not responsible for processing the MPD to discovered referenced resources. Alternatively, the Broadcaster Application could mark the entire directory, "ads/16", unused if the directory only contained the replacement ad MPD and its associated segments.

The RPC request to mark the "ads/16" directory unused would be formatted as follows:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.cache.markUnused",
  "params": {
    "elementUri": "ads/16"
  },
  "id": 42
}
```

The Receiver might respond with the following on success:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 42
}
```

Similarly, a single file could be marked unused could be requested as follows:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.cache.markUnused",  
  "params": {  
    "elementUri": "news/storyImages/photo12.png"  
  },  
  "id": 42  
}
```

The Receiver might respond with the following on success:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {},  
  "id": 42  
}
```

Standard HTTP failure codes shall be used to indicate issues with the formation of the URI and that the file or directory referenced could not be marked as unused. If an element is successfully marked as unused, future attempts to access that element have indeterminate results in that some Receivers may not have made the element unavailable and respond positively to the request while others may immediately respond with an error status.

9.10 Content Recovery APIs

9.10.1 Query Content Recovery State API

A Broadcaster Application may wish to know whether it is being managed using content recovery via watermarking and/or fingerprinting as specified in A/336 [3]. This allows the Broadcaster Application to offer different functionality in content recovery scenarios than may be offered when broadcast signaling is present and, in content recovery scenarios, it allows the application to identify the presence of modifications that may be introduced by an upstream device (such as a Set-Top Box (STB)) as discussed in Annex A of A/336 [3] on an ongoing basis during its execution and alter its behavior accordingly.

The Query Content Recovery State API shall be defined as follows:

method: "org.atsc.query.contentRecoveryState"
params: Omitted

Response:

result: A JSON object containing four key/value pairs.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "audioWatermark": {
      "type": "integer",
      "minimum": 0, "maximum": 2
    },
    "videoWatermark": {
      "type": "integer",
      "minimum": 0, "maximum": 2
    },
    "audioFingerprint": {
      "type": "integer",
      "minimum": 0, "maximum": 2
    },
    "videoFingerprint": {
      "type": "integer",
      "minimum": 0, "maximum": 2
    }
  }
}

```

audioWatermark: This integer value shall indicate one of the following states of audio watermark detection:

- 0: if the Receiver is not employing both audio watermark detection and application signaling recovered as specified in A/336 for application management;
- 1: if the Receiver is employing both audio watermark detection and application signaling recovered as specified in A/336 for application management and the Receiver is not currently detecting a VP1 Audio Watermark Segment as defined in A/336;
- 2: if the Receiver is employing both audio watermark detection and application signaling recovered as specified in A/336 for application management and the Receiver is currently detecting a VP1 Audio Watermark Segment as defined in A/336.

videoWatermark: This integer value shall indicate one of the following states of video watermark detection:

- 0: if the Receiver is not employing both video watermark detection and application signaling recovered as specified in A/336 for application management;
- 1: if the Receiver is employing both video watermark detection and application signaling recovered as specified in A/336 for application management and the Receiver is not currently detecting a VP1 Video Watermark Segment or any other video watermark message as defined in A/336;
- 2: if the Receiver is employing both video watermark detection and application signaling recovered as specified in A/336 for application management and the Receiver is currently detecting a VP1 Video Watermark Segment or any other video watermark message as defined in A/336.

audioFingerprint: This integer value shall indicate one of the following states of audio fingerprint recognition:

- 0: if the Receiver is not employing both audio fingerprint recognition and application signaling recovered as specified in A/336 for application management;
- 1: if the Receiver is employing both audio fingerprint recognition and application signaling recovered as specified in A/336 and the Receiver is not currently recognizing an audio fingerprint;
- 2: if the Receiver is employing both audio fingerprint recognition and application signaling recovered as specified in A/336 and the Receiver is currently recognizing an audio fingerprint.

videoFingerprint: This integer value shall indicate one of the following states of video fingerprint recognition:

- 0: if the Receiver is not employing both video fingerprint recognition and application signaling recovered as specified in A/336;
- 1: if the Receiver is employing both video fingerprint recognition and application signaling recovered as specified in A/336 and the Receiver is not currently recognizing a video fingerprint;
- 2: if the Receiver is employing both video fingerprint recognition and application signaling recovered as specified in A/336 and the Receiver is currently recognizing a video fingerprint.

If a key/value pair is absent in the result, it indicates that the value of the key/value pair is 0 (i.e., the associated capability is not supported by the Receiver).

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.contentRecoveryState",
  "id": 122
}
```

If the Receiver supports application management using application signaling recovered from both audio and video watermarks as specified in A/336 and both are currently being detected, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "audioWatermark": 2,
    "videoWatermark": 2
  },
  "id": 122
}
```

9.10.2 Query Display Override API

A Broadcaster Application may wish to know if the Receiver is receiving “display override” signaling obtained via watermarking (as defined in A/336 [3]) indicating that modification of the video and audio presentation should not be performed, and whether the Receiver is actively enforcing that signaling by suppressing access by the Broadcaster Application to presentation resources (“resource blocking”).

This information may be employed by the Broadcaster Application, for example, to:

- Ensure efficient utilization of Receiver and network resources (e.g. it may choose to not request resources from a broadband server when those resources cannot be presented to the user);
- Preserve an accurate representation of the user experience (e.g. to accurately report the viewability of a dynamically inserted advertisement as may be required by an ad viewability standard); or
- Comply with the requirements of the display override state, for example by halting any audio or video modification, in the event that the Receiver is not performing resource blocking.

The Query Display Override API shall be defined as follows:

method: "org.atsc.query.displayOverride"

params: Omitted

Response:

result: A JSON object containing a resourceBlocking key/value pair and a displayOverride key/value.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "resourceBlocking": {"type": "boolean"},
    "displayOverride": {"type": "boolean"}
  }
}
```

resourceBlocking: – This required Boolean value indicates if the Receiver is blocking the Broadcast Application from presenting video and audio pursuant to an active display override state as defined in A/336 [3].

displayOverride: – This required Boolean value shall be true if a display override condition is currently in effect per a video watermark Display Override Message as specified in Section 5.1.9 of A/336 [3] or per an audio watermark display override indication as specified in Sections 5.2.4 and 5.4.2 of A/336 [3]. Otherwise, the value shall be false.

If a key/value pair is absent in the result, it indicates that the value of the key/value pair is false. For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.displayOverride",
  "id": 62
}
```

And if the display override condition is currently indicated via audio watermark as specified in Section 5.2.4 of A/336 [3] and the Receiver is blocking the Broadcaster Application from presenting video and audio, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "resourceBlocking": true,
    "displayOverride": true
  },
  "id": 62
}
```

9.10.3 Query Recovered Component Info API

When content recovery via watermarking or fingerprinting is employed, it is useful for the Broadcaster Application to be able to determine which video or audio components of a service are being received by the Receiver (e.g. as a result of selection by the user on an upstream device). This can enable the Broadcaster Application to modify the on-screen placement or the language of overlaid graphics or audio to conform to the characteristics of the received component.

During content recovery via watermarking or fingerprinting, the Receiver receives component descriptors in a recovery file specified in Section 5.4.2 of A/336 [3]. This API provides a means for the Broadcaster Application to access descriptors that were recovered for components that were identified using watermarks or fingerprints.

The Query Recovered Component Info API shall be defined as follows:

method: "org.atsc.query.recoveredComponentInfo"

params: omitted.

Response:

result: A JSON object containing one or more `component` objects as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "component": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "mediaType": {
            "type": "string",
            "enum": ["audio", "video", "both"]
          },
          "componentID": {"type": "string"},
          "descriptor": {"type": "string"}
        },
        "required": ["mediaType"]
      },
      "minItems": 1
    },
    "required": ["component"]
  }
}
```

`mediaType`, `componentID` and `descriptor` are the data values associated with the media components received by the Receiver, as given in the fields of the same name in a `componentDescription` element of a recovery file as specified in Table 5.29 of A/336 [3].

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.recoveredComponentInfo",
  "id": 39
}
```

If a `componentDescription` element of the recovery file lists an audio component with the `componentID` value “1” and the `descriptor` value “component descriptor string 1”, and a video component with the `componentID` value “2” and the `descriptor` value “component descriptor string 2” associated with the components received by the Receiver, the Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"component": [
    {
      "mediaType": "audio",
      "componentID": "1",
      "descriptor": "component descriptor string 1"
    },
    {
      "mediaType": "video",
      "componentID": "2",
      "descriptor": "component descriptor string 2"
    }
  ]},
  "id": 39
}
```

9.10.4 Content Recovery State Change Notification API

The Content Recovery State Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the content recovery state as defined in Query Content Recovery State API in Section 9.10.1 changes from one state to another different state in which at least one property value changes.

The Content Recovery State Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object containing four key/value pairs representing the new content recovery state.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "string",
      "enum": ["contentRecoveryStateChange"]
    },
    "audioWatermark": {
      "type": "integer",
      "minimum": 0,
      "maximum": 2
    },
    "videoWatermark": {
      "type": "integer",
      "minimum": 0,
      "maximum": 2
    },
    "audioFingerprint": {
      "type": "integer",
      "minimum": 0,
      "maximum": 2
    },
    "videoFingerprint": {
      "type": "integer",
      "minimum": 0,
      "maximum": 2
    }
  },
  "required": ["msgType"]
}
```

audioWatermark, videoWatermark, audioFingerprint and videoFingerprint are defined in the Query Content Recovery State API in Section 9.10.1.

If a key/value pair is absent in the `params`, it indicates that the value of the key/value pair is 0.

For example, if the user changes from a non-watermarked service to a new service marked with both audio and video watermarks and both audio and video watermarks are detected and being used for content recovery in the Receiver, the Receiver notifies the Broadcaster Application the content recovery state change as shown below:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "contentRecoveryStateChange",
    "audioWatermark": 2,
    "videoWatermark": 2
  }
}
```

9.10.5 Display Override Change Notification API

The Display Override Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the display override state or resource blocking state as defined in Query Display Override API in Section 9.10.2 changes from one state to another different state.

The Display Override Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key/value named `msgType` with value "displayOverrideChange", a `resourceBlocking` key/value pair, and a `displayOverride` key/value pair.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "string",
      "enum": ["displayOverrideChange"]
    },
    "resourceBlocking": {"type": "boolean"},
    "displayOverride": {"type": "boolean"}
  },
  "required": ["msgType"]
}
```

`resourceBlocking` and `displayOverride` are defined in Query Display Override API in Section 9.10.2.

If a key/value pair is absent in the `params`, it indicates that the value of the key/value pair is false.

For example, if the display override state changes from inactive to active and the Receiver is blocking the currently executing Broadcaster Application from presenting video and audio, the Receiver notifies the Broadcaster Application the Display Override change as shown below:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "displayOverrideChange",
    "resourceBlocking": true,
    "displayOverride": true
  }
}
```

9.10.6 Recovered Component Info Change Notification API

The Recovered Component Info Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the video or audio components of a service being received by the Receiver changes (e.g. as a result of selection by the user on an upstream device).

The Recovered Component Info Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting a key named `msgType` with value "recoveredComponentInfoChange" and two key/value pairs describing the new component of the service.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "string",
      "enum": ["recoveredComponentInfoChange"]
    },
    "mediaType": {
      "type": "string",
      "enum": ["audio", "video", "both"]
    },
    "componentID": {"type": "string"},
    "descriptor": {"type": "string"}
  },
  "required": ["msgType", "mediaType"]
}
```

`mediaType`, `componentID` and `descriptor` are defined in the Query Recovered Component Info API in Section 9.10.3.

For example, if the user at an upstream device of the Receiver changed from Spanish to English audio track described by the `componentID` value "1" and `descriptor` value "component description string 3", the Receiver notifies the Broadcaster Application the recovered component changed as shown below:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "recoveredComponentInfoChange",
    "mediaType": "audio",
    "componentID": "1",
    "descriptor": "component description string 3"
  }
}
```

9.11 Filter Codes APIs

The Receiver may use Filter Codes to selectively download NRT data files by comparing the stored Filter Codes with the Filter Codes associated with the NRT data files in the EFDT.

Note that a Broadcaster Application cannot predict for how long a Receiver will retain a set of Filter Codes. Thus, it is recommended and expected that the Set Filter Codes API always provides *all* the Filter Codes that apply to a given Receiver. In this way, the same API is used to create a new Filter Codes definition and to update a Filter Codes definition. Each Set Filter Codes operation

completely replaces the old definition in the Receiver. It is further possible to delete previously defined Filter Code values by setting a null set of Filter Codes using the Set Filter Codes API.

If the Receiver has downloaded one or more files based on a particular Filter Code value and a new Filter Codes definition is received that does not include that value, the Receiver can be expected to use this information as a caching hint, and thus may choose to delete those files.

9.11.1 Get Filter Codes API

The Get Filter Codes API can be used by a Receiver to discover filtering terms that the Receiver can use to determine which files to download on behalf of an application.

The Get Filter Codes API shall be defined as follows:

method: "org.atsc.getFilterCodes"

params: Omitted.

Response:

result: A JSON object containing an object with key/value pairs as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "filters": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "filterCode": {"type": "integer"},
          "expires": {"type": "string", "format": "xs:dateTime"},
        },
        "required": ["filterCode"]
      }
    },
  },
  "required": ["filters"]
}
```

filters – A required array of objects, each containing a required Filter Code and an optional expiration time.

filterCode – A required unsigned integer associated with personalization categories as determined by the broadcaster. It is the broadcaster's responsibility to maintain a scope of uniqueness of Filter Codes to be within an `AppContextID`.

expires – This string shall be represented by the `xs:dateTime` XML data type as defined in the W3C XML Schema [28] to indicate the expiry of the Filter Code. Filter Codes are not expected to be used after expiry. If the value is omitted, then no expiration is indicated.

For example, the Receiver can request Filter Codes from a Broadcaster Application by issuing:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.getFilterCodes",
  "id": 57
}
```

Upon success, the Broadcaster Application might respond:

```
--> {
  "jsonrpc": "2.0",
  "result": {
    "filters": [
      { "filterCode": 101, "expires": "2016-07-17T09:30:47" },
      { "filterCode": 102, "expires": "2016-07-17T09:30:47" },
      { "filterCode": 103 } ]
  },
  "id": 57
}
```

9.11.2 Set Filter Codes API

The Set Filter Codes API can be issued by a Broadcaster Application to notify the Receiver to store the specified Filter Codes.

The Set Filter Codes API shall be defined as follows:

method: "org.atsc.setFilterCodes"

params: A JSON object containing an object with key/value pairs as defined below.

params JSON Schema:

```
{
  "type": "object",
  "properties": { "filters": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "filterCode": { "type": "integer" },
        "expires": { "type": "string", "format": "xs:dateTime" }
      },
      "required": [ "filterCode" ]
    }
  },
  "required": [ "filters" ]
}
```

filterCode – An unsigned integer associated with personalization categories as determined by the broadcaster. It is the broadcaster's responsibility to maintain a scope of uniqueness of Filter Codes to be within an AppContextID.

expires – This string shall be represented by the xs:dateTime XML data type as defined in the W3C XML Schema [28] to indicate the expiry of a Filter Code. Filter Codes are not expected to be used after expiry. If the value is omitted, then no expiration is indicated.

In the following example, the Broadcaster Application provides three filter codes for the Receiver to use:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.setFilterCodes",
  "params": {
    "filters": [
      {"filterCode": 101, "expires": "2016-07-17T09:30:47"},
      {"filterCode": 102, "expires": "2016-07-17T09:30:47"},
      {"filterCode": 103}]
    },
  "id": 57
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 57
}
```

9.12 Keys APIs

The APIs in this section allow the Broadcaster Application, with the Receiver's permission, to access certain specified remote-control keys.

9.12.1 Request Keys API

A Broadcaster Application can request to receive keypresses that are typically used and processed by Receivers. For example, numeric keypresses on the remote control are typically used by the underlying Receiver to directly tune to a specific channel. However, the Broadcaster Application may wish to present a data entry UI to accept numeric data from the user in order to perform a specific action or to solicit input from the viewer. In this case, the Broadcaster Application can request the Receiver to temporarily re-route numeric keypresses to itself. Based on the Receiver manufacturer, the Receiver may reject this request, in which case, the Broadcaster Application may choose to display a soft keyboard on the TV screen or resort to using other types of device input.

The Request Keys API shall be defined as follows:

method: "org.atsc.request.keys"

params: a JSON object indicating the type of keys requested

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "keys": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": ["Numeric", <other>]
      }
    },
    "required": ["keys"]
  }
}

```

keys – This parameter shall be an array of strings, each representing a particular remote-control key or type of key the Broadcaster Application would like the Receiver to forward.

- "Numeric" – Indicates the numeric keys 0-9.
- <other> – Indicates any of the strings in W3C “UI Events KeyboardEvent key Values,” Section 3.18, Media Controller Keys [25].

Response:

The Receiver shall respond with a JSON-RPC response object including a result object. The result object includes a string array indicating the keys for which the request was successful. It can be assumed that any requested keys that are not included in the "accepted" array were not accepted.

result JSON schema:

```

{
  "type": "object",
  "properties": {
    "accepted": {
      "type": "array",
      "items": {"type": "string"}
    },
    "required": ["accepted"]
  }
}

```

For example, if the Broadcaster Application requests receipt of numeric keys and Channel Up and Channel Down arrows, it can issue this request to the Receiver:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.request.keys ",
  "params": {"keys": ["Numeric", "ChannelUp", "ChannelDown"]},
  "id": 43
}

```

If the Receiver grants the numeric keypresses but not the Channel Up and Channel Down keypresses, the Receiver could respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"accepted": ["Numeric"]},
  "id": 43
}
```

9.12.2 Relinquish Keys API

A Broadcaster Application can relinquish previous requests for keypresses. This would be used after a request made via the Request Keys API (Section 9.12.1) to return the handling of keypresses to a normal state.

The Relinquish Keys API shall be defined as follows:

method: "org.atsc.relinquish.keys"

params: optional, includes optional "keys" string array, indicating which keys to relinquish. If absent (or if "keys" is absent or is an empty string), all keys will be relinquished.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "keys": {
      "type": "array",
      "items": {"type": "string", "enum": ["Numeric", <other>]}
    }
  }
}
```

keys – This parameter is an array of strings, each representing a particular remote-control key or type of key the Broadcaster Application would like the Receiver to relinquish. If the keys parameter is not provided, then all keys requested for forwarding shall be relinquished.

- "Numeric" – Indicates the numeric keys 0-9.
- <other> – Indicates any of the strings in W3C “UI Events KeyboardEvent key Values,” Section 3.18, Media Controller Keys [25]].

Response:

The Receiver shall respond with a JSON-RPC response object with a null result object.

For example, the Broadcaster Application can issue this request to the Receiver:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.relinquish.keys",
  "params": {
    "keys": ["ChannelUp", "ChannelDown"]
  },
  "id": 44}
```


The Receiver might respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 44
}
```

9.12.3 Request Keys Timeout

To help to avoid application misbehavior, the Receiver may force keypress requests to be relinquished after a certain amount of time, defined by each manufacturer. This is referred to as a request key timeout. Prior to the request key timeout, the Receiver sends a warning notification to the Broadcaster Application to provide the application time to respond to the timeout. The Broadcaster Application may, at this point, choose to issue another Request Keys API or it may allow the key request(s) to time out. If another request key API is issued, it may or may not be accepted by the Receiver.

The Request Key Timeout Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object including a key named `msgType` with the value "requestKeyTimeout". The params object also includes a timeout array that identifies each key that is nearing timeout along with the number of seconds until timeout occurs.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["requestKeyTimeout"]},
    "timeout": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "key": {"type": "string"},
          "time": {"type": "integer"}
        },
        "required": ["key", "time"]
      },
      "minItems": 1
    }
  },
  "required": ["msgType", "timeout"]
}
```

key – This parameter is a string representing a particular remote-control key or type of key about which the Receiver wishes to notify the Broadcaster Application.

- "Numeric" – Indicates the numeric keys 0-9.
- <other> – Indicates any of the strings in W3C "UI Events KeyboardEvent key Values," Section 3.18, Media Controller Keys [25].

time – An integer number of seconds indicating the timeout for the given key or type of key.

For example, the Receiver may send the following notification to indicate that Channel Up and Channel Down key requests will time out in 3 seconds:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "requestKeyTimeout",
    "timeout": [
      {"key": "ChannelUp", "time": 3},
      {"key": "ChannelDown", "time": 3}
    ]
  }
}
```

9.13 Query Device Info API

The Query Device Info API provides an interface between a Broadcaster Application and the Receiver to retrieve device-specific information. It is a generic conduit between the Receiver and the Broadcaster Application to provide basic device information including make and model of device, along with optional additional key/value pair information about the device. The format and definition of the optional additional key/value pairs are manufacturer-specific and not specified here. Specific parameters may be defined as part of a business relationship between a broadcaster and a device manufacturer.

The two unique ID parameters (`deviceId`, `advertisingId`) in the response below are expected to be initialized once by the Receiver to afford long-term persistence across all Services and Receiver power cycles and be provided by the Receiver depending on authorization granted by the user. The ability to authorize the disclosure of either the unique `deviceId` or `advertisingId` parameters, either per broadcaster, per Broadcaster Application or as a whole, affords increased privacy to the user.

The Query Device Info API request `params` object is optional. If `params` is omitted (or if `deviceInfoProperties` is omitted or is an empty array), the Receiver shall respond with only the device make and model. The Broadcaster Application can then use the device make and model to determine which additional properties to query. The `deviceInfoProperties` is an array of desired properties, and the Receiver provides the values of these properties in the response.

The Query Device Info API shall be defined as follows:

method: "org.atsc.query.deviceInfo"

params: An optional JSON object.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "deviceInfoProperties": {
      "type": "array",
      "items": {"type": "string"}
    }
  }
}
```

`deviceInfoProperties` – This parameter is an array of strings, each representing a particular aspect of the device about which the Broadcaster Application is interested.

Response:

`result`: a JSON object containing the device make and model and optionally a request for additional information about a given device make/model.

`result` JSON Schema:

```
{
  "type": "object",
  "properties": {
    "deviceMake": {"type": "string"},
    "deviceModel": {"type": "string"},
    "deviceInput": {"type": "object"},
    "deviceInfo": {"type": "object"},
    "deviceId": {"type": "string"},
    "advertisingId": {"type": "string"},
    "deviceCapabilities": {"type": "string"}
  },
  "required": ["deviceMake", "deviceModel", "deviceInput"]
}
```

`deviceMake`: – This required string indicates the manufacturer of the Receiver.

`deviceModel`: – This required string indicates the model of the Receiver.

`deviceInput`: – This required object indicates the user input name and codes of the Receiver user interface. It is a collection of input key/value pairs where the key is the user input name and the value is the associated integer code. The minimum set of user input names is listed in Table 9.4.

`deviceInfo`: – This optional object includes key/value pairs. The `deviceInfo` is included in the response if the request included one or more `deviceInfoProperties` strings corresponding to information the Receiver can supply.

`deviceId`: – This optional string returns a globally unique UUID as defined in RFC 4122 [22] using the urn:uuid syntax when authorized for the particular Broadcaster Application. When absent, it is not supported by the Receiver. When present and all zeros, the value of the user setting to provide this identifier is disabled for the given service indicating that the user has explicitly not authorized provision of the value.

`advertisingId`: – This optional string returns a globally unique UUID as defined in RFC 4122 [22] using the urn:uuid syntax when authorized for the particular Broadcaster Application. When absent, it is not supported by the Receiver. When present and all zeros, the value of the user setting to provide this identifier is disabled for the given service indicating that the user has explicitly not authorized provision of the value.

`deviceCapabilities`: – This optional string describes the capabilities, or features, of the receiver and shall conform to A/332, Section 5.2.2.3.3.2 “Device Capabilities Syntax and Semantics” [2]. This is, in essence, a standardized version of the `deviceInfo` string.

Although it is possible for the Receiver and the Broadcaster Application to negotiate receiving additional key events using the WebSocket APIs provided in Section 9.12, the User Agent shall

provide the key code values marked as ‘Required’ in Table 9.4 to the Broadcaster Application when input by the user.

Table 9.4 Input Key Code Values

Function	Support	Key Code Value	Description
ArrowUp	Required	38	Up arrow key or equivalent
ArrowDown	Required	40	Down arrow key or equivalent
ArrowRight	Required	39	Right arrow key or equivalent
ArrowLeft	Required	37	Left arrow key or equivalent
Select	Required	13	Indicates to the Broadcaster Application that the user wishes to select
Back	Required	461	Indicates to the Broadcaster Application that the user wishes to go back a step
Home	Optional	503	Indicates to the Broadcaster Application that the user wishes to go back to the home/start page

ArrowUp – This key event is sent when the user triggers the up arrow directional key on the remote control or through some equivalent action. The User Agent shall generate the keycode value of 38 in response to a user up arrow input or equivalent.

ArrowDown – This key event is sent when the user triggers the down arrow directional key on the remote control or through some equivalent action. The User Agent shall generate the keycode of 40 in response to a user down arrow input or equivalent.

ArrowRight – This key event is sent when the user triggers the right arrow directional key on the remote control or through some equivalent action. The User Agent shall provide the keycode of 39 in response to a user right arrow input or equivalent.

ArrowLeft – This key event is sent when the user triggers the left arrow key on the remote control or through some equivalent action. The User Agent shall provide the keycode of 37 in response to a user right arrow input or equivalent.

Select – This key event is sent when the user triggers a remote-control key signaling selection. Examples of common remote-control keys that may provide a selection input include ‘OK’, ‘Enter’, ‘Select’ and ‘Return’ keys. The Select event indicates that the user wishes the Broadcaster Application to take an action based on the current focus. For example, if an icon is highlighted, the Select key code indicates that the user wishes to engage with the associated function. The User Agent shall provide the keycode value of 13 when the user provides a selection input or equivalent.

Back – This key event is sent when the user triggers a remote-control key signaling the desire to move back a step, depending on the current Broadcaster Application state. For example, the Back event may indicate to the Broadcaster Application that activities on the current display are completed and the user wishes to return to a previous display. The User Agent shall provide the keycode value of 461 when the user provides a Back input or equivalent.

Home – This key event is sent when the user triggers a remote-control key signaling the desire to move to the home page. It is up to the Broadcaster Application on how to use this remote-control key event, if it is made available. If the Receiver makes this key available, the User Agent shall provide the keycode value of 503 when the user provides a Home input or equivalent.

For example, a query from the Broadcaster Application may look like this:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.query.deviceInfo",  
  "id": 91  
}
```

A Receiver manufactured by the “Acme” company with model number “A300” might respond with:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {  
    "deviceMake": "Acme",  
    "deviceModel": "A300",  
    "deviceInput": {  
      "ArrowUp": 38,  
      "ArrowDown": 40,  
      "ArrowRight": 39,  
      "ArrowLeft": 37,  
      "Select": 13,  
      "Back": 461  
    }  
  },  
  "id": 92  
}
```

If the Broadcaster Application recognizes this make and model, it might know that additional information about this Receiver may be retrieved by indicating specific device properties in the request. Note that not all Receivers would be expected to recognize the `deviceInfoProperties` strings given in this example:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.query.deviceInfo",  
  "params": {"deviceInfoProperties": ["numberOfTuners", "yearOfMfr"]},  
  "id": 93  
}
```

This Receiver might respond with:

```

<-- {
  "jsonrpc": "2.0",
  "result": {
    "deviceMake": "Acme",
    "deviceModel": "A300",
    "deviceInput": {
      "ArrowUp": 38,
      "ArrowDown": 40,
      "ArrowRight": 39,
      "ArrowLeft": 37,
      "Select": 13,
      "Back": 461
    },
    "deviceInfo": {
      "numberOfTuners": 1,
      "yearOfMfr": 2017
    }
  },
  "id": 93
}

```

9.14 RMP Content Synchronization APIs

9.14.1 Query RMP Media Time API

A Broadcaster Application may wish to know the current presentation time of the broadcast content being presented by the Receiver. This enables the Broadcaster Application to present supplemental content that is synchronized to the content. For example, a Broadcaster Application may display a graphical overlay at a particular moment in a program.

The Query RMP Media Time API shall be defined as follows:

method: "org.atsc.query.rmpMediaTime"

params: Omitted

Response:

result: A JSON object containing two key/value pairs.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "currentTime": {"type": "number", "minimum": 0},
    "startDate": {"type": "string", "format": "xs:dateTime"}
  },
  "required": ["currentTime"]
}

```

currentTime – This required floating-point number value shall represent the current presentation time of the content being presented by the RMP, expressed as an offset, in seconds, to **startDate**. It shall have the same meaning as the **@currentTime** attribute of the **HTMLMediaElement** given in [7] that represents the official playback position of the content.

startDate – This optional **xs:dateTime** value represents an absolute time reference for the start (i.e., the zero time) of the media timeline of the content being presented by the RMP. It has the same

meaning as the `@startDate` attribute of the `HTMLMediaElement` given in [7]. The `xs:dateTime` XML data type shall be formatted as defined in the W3C XML Schema [28].

When the RMP is presenting content compliant with [32], the following requirements apply to the reported values of `startDate` and `currentTime`:

- The value of `startDate` represents the sum of `MPD@availabilityStartTime` in the MPD that was in use by the RMP when it began playing or recording the presentation and the time offset on the DASH Media Presentation timeline at which the RMP began playing or recording the presentation. When content delivered via broadband allows the RMP to seek to a position in the presentation earlier than the time at which RMP began playing or recording the content (e.g. live time-shift), the time offset on the DASH Media Presentation timeline shall be the earliest seek-able time offset in the content. Note that the media format of the recorded content is receiver specific.
- When recorded content is being presented, both the `startDate` and the `currentTime` values shall have the same respective values as the `startDate` and the `currentTime` values applied during presentation of the live version of the recorded content.

If no explicit date and time is available for the content being present (e.g., the downloaded content), `startDate` shall be absent in the response. Otherwise, `startDate` shall be present in the response.

For example, if the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.rmpMediaTime",
  "id": 61
}
```

The Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": { "currentTime": 3600.033,
             "startDate": "2019-01-01T23:59:59.590Z"
          },
  "id": 61
}
```

9.14.2 Query RMP Wall Clock API

A Broadcaster Application may wish to know the wall clock time, which is synchronized to the PTP Time signaled in the broadcast physical layer protocol. The wall clock can be used to query the current UTC time instead of the JavaScript API, e.g. `Date.now()` of the User Agent.

The Query RMP Wall Clock Time API shall be defined as follows:

method: "org.atsc.query.rmpWallClockTime"

params: Omitted

Response:

result: A JSON object containing one key/value pair.

result JSON Schema:

```

{
  "type": "object",
  "properties": {
    "wallClock": {"type": "string", "format": "xs:dateTime"}
  },
  "required": ["wallClock"]
}

```

wallClock – This required xs:dateTime value shall represent the wall clock time, which is synchronized to the PTP Time signaled in the broadcast physical layer protocol. The xs:dateTime XML data type shall be formatted as defined in the W3C XML Schema [28].

For example, if the Broadcaster Application makes a query:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.rmpWallClockTime",
  "id": 62
}

```

The Receiver might respond:

```

<-- {
  "jsonrpc": "2.0",
  "result": {"wallClock": "2019-01-01T23:59:56.320Z"},
  "id": 62
}

```

9.14.3 Query RMP Playback State API

A Broadcaster Application may wish to know the playback state of the content being presented or prepared for presentation by RMP. This allows the application to make adjustments in presenting supplemental content based on playback state of the content. For example, the application may suspend presentation of supplemental content if playback of the presentation is paused due to content buffer underflow (“buffering”) or user input or stopped due to reaching the end of a VOD program stream.

The Query RMP Playback State API shall be defined as follows:

method: "org.atsc.query.rmpPlaybackState"

params: Omitted

Response:

result: A JSON object containing a `playbackState` key/value pair.

result JSON Schema:

```

{
  "type": "object",
  "properties": {"playbackState": {"type": "integer", "minimum": -1, "maximum": 3}},
  "required": ["playbackState"]
}

```


`playbackState` –This integer value shall indicate one of the following playback states of the RMP:

- -1 if the content is initializing, connecting and the state cannot be determined, for example there is a time window between changing the channel, accessing the SLT, MPD and initializing the channel where the state is unclear;
- 0 if the content is actively playing, and if encrypted, there are necessarily also valid DRM licenses and the CDM is decrypting the content;
- 1 if the playback is paused for any reason and has not ended (e.g. seeking or stalled, paused for user interaction, waiting for user input, stopped due to errors);
- 2 if the playback has ended (e.g. the end of the content is reached).
- 3 if the content is encrypted and not viewable (i.e. there are no valid keys and the CDM is not decrypting the content).

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.rmpPlaybackState",
  "id": 63
}
```

And if the RMP is playing back content, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"playbackState": 0},
  "id": 63
}
```

9.14.4 Query RMP Playback Rate API

A Broadcaster Application may wish to know the speed at which the content is being presented by the RMP. This allows the application to make adjustments in presenting supplemental content based on playback speed of the content. For example, the application may choose to suspend presentation of supplemental content during trick-play mode.

The Query RMP Playback Rate API shall be defined as follows:

method: "org.atsc.query.rmpPlaybackRate"

params: Omitted

Response:

result: A JSON object containing a `playbackRate` key/value pair.

result JSON Schema:

```
{
  "type": "object",
  "properties": {"playbackRate": {"type": "number"}},
  "required": ["playbackRate"]
}
```

`playbackRate` – This required floating-point value indicates the playback speed of the content currently being presented by RMP with the same meaning as the attribute `playbackRate` of `HTMLMediaElement` as given in [7].

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.rmpPlaybackRate",
  "id": 65
}
```

And if the RMP is playing back content at the normal speed, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"playbackRate": 1.0},
  "id": 65
}
```

9.14.5 RMP Media Time Change Notification API

The RMP Media Time Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the current playback position of the content being presented by the RMP changed. This API has the same meaning functionally as the `timeupdate` event of `HTMLMediaElement` as given in [7]. The Receiver notifies the Broadcaster Application using this API when the official playback position of the RMP changes as part of normal or trick playback.

The RMP Media Time Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object containing three key/value pairs.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["rmpMediaTimeChange"]},
    "currentTime": {"type": "number", "minimum": 0},
    "startDate": {"type": "string", "format": "xs:dateTime"}
  },
  "required": ["msgType", "currentTime"]
}
```

`currentTime` and `startDate` are defined in Query RMP Media Time API in Section 9.14.1. If the `startDate` key/value is absent in the `params`, it indicates that the value of the key/value pair is unchanged.

For example, the Receiver may notify the application the media time change every 250 to 500 msec. during the normal playback of the current service. A notification provided shortly after the previous notification which contained a `currentTime` value 3600.033 might be:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "rmpMediaTimeChange",
    "currentTime": 3600.283
  }
}
```

9.14.6 RMP Playback State Change Notification API

The RMP Playback State Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the playback state of the RMP as defined in Query RMP Playback State API in Section 9.14.3 changes from one value to another different value.

The RMP Playback State Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of two key/value pairs.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["rmpPlaybackStateChange"]},
    "playbackState": {"type": "integer", "minimum": -1, "maximum": 3}
  },
  "required": ["msgType", "playbackState"]
}
```

playbackState – This integer value shall indicate the new playback state that is one of the playback states defined in Query Playback State API in Section 9.14.3.

For example, if the user at the Receiver pauses the playback of the time-shift broadcast content, the Receiver notifies the Broadcaster Application the playback state as shown below:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "rmpPlaybackStateChange",
    "playbackState": 1
  }
}
```

9.14.7 RMP Playback Rate Change Notification API

The RMP Playback Rate Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if the playback speed as defined in Query RMP Playback Rate API in Section 9.14.4 changes from one value to another different value.

The RMP Playback Rate Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of two key/value pairs.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["rmpPlaybackRateChange"]},
    "playbackRate": {"type": "number"}
  },
  "required": ["msgType", "playbackRate"]
}

```

`playbackRate` shall be as defined in Query RMP Playback Rate API in Section 9.14.4.

For example, if the user at the Receiver performs fast-forward playback of the time-shift content at 2 times normal playback speed, the Receiver notifies the Broadcaster Application the playback speed changes as shown below:

```

<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "rmpPlaybackRateChange",
    "playbackRate": 2
  }
}

```

9.15 DRM APIs

The APIs in this section can be used by the Broadcaster Application to support the RMP or AMP handling of encrypted content. Two generic APIs are defined. A “notification” API is used by the Receiver to pass a message associated with an identified DRM System to the Broadcaster Application. An “operation” API is used by the Broadcaster Application to pass a message associated with an identified DRM System to the Receiver. These APIs support the needs of both the AMP and the RMP.

9.15.1 DRM Notification API

The DRM Notification API may be issued by the Receiver to the Broadcaster Application in order to deliver a DRM-related notification. A Broadcaster Application which receives this notification can use the DRM Operation API, defined in Section 9.15.2, to exchange a message with the Receiver’s underlying content protection system, ultimately resulting in the delivery of the license/key required by the RMP or AMP for decryption of protected content.

The DRM Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object containing three key/value pairs.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["DRM"]},
    "systemId": {"type": "string"},
    "message": {"type": "array", "items": {"type": "object"}},
  },
  "required": ["msgType", "systemId", "message"]
}

```

msgType – This required string shall be set to "DRM" to identify this notification API.

systemId – This string shall be set to a DRM system identifier, `@schemeIdUri`, as defined in the DASH-IF IOP, section 7.6 [32]. For example, the UUID string value "1077efec-c0b2-4d02-ace3-3c1e52e2fb4b" corresponds to the common system ID of W3C EME.

message – This shall be the message passed from the content protection system formatted as an array of JSON objects.

The DRM Notification API may be used by the Receiver to notify the Broadcaster Application that a particular DRM license object that had been requested via the DRM Operation API (Section 9.15.2) has been retrieved from the broadcast. In the following example, the `message` object includes key/value pairs providing the URI of the requested license file, and the contents of the retrieved file. The key/value pairs within the `message` object (in this example, the keys are `licenseUri` and `licenseData`) are as specified by the DRM system identified in the `systemId` key:

```

<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "DRM",
    "systemId": "urn:uuid:1077efec-c0b2-4d02-ace3-3c1e52e2fb4b",
    "message": [{ "licenseUri": "mybroadcast/myDRM/mysubID/license.dat",
                  "licenseData": "ZkxhQwAAACIQABAAAAUJABtA..." }]
  }
}

```

As another example, the Receiver may notify the application if the base layer of video content is playing properly but a decryption key is needed for the enhancement layer of a spatial scalable video codec, e.g. SHVC.

```

<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "DRM",
    "systemId": "urn:uuid:1077efec-c0b2-4d02-ace3-3c1e52e2fb4b",
    "message": [{ "<proprietary>": "<proprietary>", {...} } ]
  }
}

```

It is out of scope of the present specification how a Broadcaster Application interacts with a license server and the exact format of messages passed from the content protection system to the Broadcaster Application using this DRM Notification API. The format of messages passed across the interface defined in this API depends on the content protection system used by the broadcaster.

9.15.2 DRM Operation API

The DRM Operation API can be issued by a Broadcaster Application to pass a message to the Receiver in order to play protected content as defined by Section 5.7 of [6]. This API can be used along with the DRM Notification API as defined in Section 9.15.1 which is issued by the Receiver to notify a message to a Broadcaster application in order to inform information about content protection.

Similar to W3C EME [24] which is an extension of the HTML5 media element, a Broadcaster Application can communicate with a license server and pass messages for license/key exchange to the underlying content protection system via this API. These APIs are simpler than W3C EME since the APIs only provide a message exchange mechanism and the content of the messages conveyed in the API, including any control such as installing/updating/removing licenses, are specific to an underlying content protection system and thus not specified here. Note that a Broadcaster Application needs to know the sequence of interactions with a broadcaster's web server and a license server and also the procedure for exchange of messages with the underlying content protection system of the Receiver.

The DRM Operation API shall be defined as follows:

method: "org.atsc.drmOperation"

params: A JSON object consisting of `systemId` and message properties as follows.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "systemId": {"type": "string"},
    "service": {"type": "string"},
    "message": {"type": "array", "items": {"type": "object"}}
  },
  "required": ["systemId", "service", "message"]
}
```

`systemId` – This string shall be set to a DRM system identifier, `@schemeIdUri`, as defined in the DASH-IF, section 7.6 [39].

`service` – This required string shall indicate the globally unique Service ID associated with the currently selected service as given in the SLT in `SLT.service@globalServiceID` to which this DRM message is associated.

`message` – This shall be a message specific to the content protection system in use formatted as an array of JSON objects.

Response:

result: The following JSON object is returned upon success.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "message": {"type": "array", "items": {"type": "object"}}
  }
}
```

error: The following error codes may be returned:

- -14: The specified content protection system is not supported by the Receiver.

message – This shall be the response to the command formatted as an array of JSON objects specific to the content protection system in use.

In the case of the AMP, when the DRM Operation API is used by the Broadcaster Application to ask the Receiver to fetch a particular DRM license file from the broadcast Entitlement Management service, the message object may include key/value pairs to specify the Content-Location of the file, e.g. `mybroadcast/myDRM/mysubID/license.dat` in the following example:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.drmOperation",
  "params": {
    "systemId": "urn:uuid:1077efec-c0b2-4d02-ace3-3c1e52e2fb4b",
    "service": "https://doi.org/10.5239/8A23-2B0",
    "message": {
      "operation": "licenseMessageRequest",
      "licenseUri": "mybroadcast/myDRM/mysubID/license.dat"
    }
  },
  "id": 101
}
```

Upon successfully accepting the request to fetch this file, the Receiver may respond with:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 101
}
```

As another example, the Broadcaster Application can pass a proprietary message for the DRM system corresponding to the UUID `1077efec-c0b2-4d02-ace3-3c1e52e2fb4b` by issuing a command to the Receiver as follows:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.drmOperation",
  "params": {
    "systemId": "urn:uuid:1077efec-c0b2-4d02-ace3-3c1e52e2fb4b",
    "message": [{"<proprietary>":"<proprietary>"},{...}]
  },
  "id": 104
}
```

If the Receiver supports this DRM system, it could respond without an error:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 104
}
```

If the Receiver does not support this DRM system, it could respond with an error message:

```
<-- {
  "jsonrpc": "2.0",
  "error": { "code": -14, "message": "The specified content protection
system is not supported by the Receiver" },
  "id": 104
}
```

9.16 XLink APIs

The APIs in this section provide a mechanism allowing the Broadcaster Application to replace a DASH period marked with an XLink attribute with a period fragment referencing alternate content. There are two APIs defined:

- A notification API allowing the Receiver to notify the Broadcaster Application that the RMP has detected an MPD period with an XLink attribute. Notifications only occur if the Broadcaster Application has subscribed to the notification (see Section 9.7.5.1). The Broadcaster Application may then provide an alternate period using the XLink Resolved API.
- The XLink Resolved API is used by the Broadcaster Application to provide a replacement MPD URL or DASH period text to be used by the RMP as an alternative to the DASH period within which the XLink was detected. It is possible for the Receiver to ignore the XLink Resolved request for a variety of reasons. Once an MPD URL or DASH period is sent to the Receiver, the Receiver shall provide timing information in the response, so the Broadcaster Application can present the appropriate user interface.

9.16.1 XLink Resolution Notification API

An XLink Resolution Notification shall be issued by the Receiver to the currently executing Broadcaster Application when the Receiver Media Player (RMP) encounters a `period@xlink:href` attribute in the form of a tag URI beginning with `tag:atsc.org,2016:xlink` in the MPD. Any XLinks not beginning with this URI are expected to be disregarded or processed in a proprietary manner. For XLinks beginning with `tag:atsc.org,2016:xlink`, the Receiver

shall notify the Broadcaster Application to resolve the XLink, if the Broadcaster Application is subscribed to the XLink Resolution Notification (see Section 9.7.5.1). The Broadcaster Application may respond by using the XLink Resolved API (see Section 9.16.2) to replace the period element in which the XLink appeared.

The XLink Resolution Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object containing the type of message, "xlinkResolution", and the XLink string detected in the DASH Period.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {
      "type": "string",
      "enum": ["xlinkResolution"]
    },
    "xlink": {
      "type": "string",
      "format": "uri"
    }
  },
  "required": ["msgType", "xlink"]
}
```

If the Receiver encounters a new MPD with the same XLink URN, the Receiver shall suppress sending any duplicate notifications associated with the XLink URN. This simplifies the request/response model described below. Consequently, XLinks inserted into MPD Periods shall contain a distinct URN.

Once an XLink Resolution notification is sent, the Broadcaster Application may decide to provide alternate content using its own criteria.

For example, the broadcast stream may contain advertisements that could be targeted at various viewers based on a variety of criteria. The time span of the default advertisement and the associated content would be described in a particular DASH Period structure that would contain an XLink attribute. To accommodate the advertisement replacement, the Broadcaster Application would subscribe to the XLink Notification API. On receipt of the Period containing the XLink, the Receiver would issue the following message:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "xlinkResolution",
    "xlink": "urn:xbc 4399FB77-3939EA47"
  }
}
```

9.16.2 XLink Resolved API

After the XLink Notification (Section 9.16.1) is received, the Broadcaster Application determines which alternate content to replace the default content with and makes a request to the Receiver using the XLink Resolved API. The Broadcaster Application provides either the MPD URL of the alternate content or the actual period text in the request. The Receiver processes the replacement MPD URL or Period text, as appropriate, to replace the default content with the alternate content. If the Receiver is not able to replace the default content, the returned disposition object shall contain the appropriate code and description indicating the reason for the failure. In either case,

timing information is provided to allow the Broadcaster Application to provide an appropriate user interface to match the playing content.

The XLink Resolved API shall be defined as follows:

method: "org.atsc.xlinkResolution"

params: A JSON object consisting of a key named `xlink` and a string representing the contents of the "xlink:href" element and either an MPD URL reference or the replacement Period text.

params JSON Schema:

```
{
  "type": "object",
  "oneOf": [
    {
      "properties": {
        "xlink": {"type": "string", "format": "uri"},
        "mpdURL": {"type": "string", "format": "uri"}
      },
      "required": ["xlink", "mpdURL"]
    },
    {
      "properties": {
        "xlink": {"type": "string", "format": "uri"},
        "period": {"type": "string", "format": "xml"}
      },
      "required": ["xlink", "period"]
    }
  ]
}
```

`xlink` – This required string shall be the XLink value from the `xlink:href` attribute in the MPD Period element. This string corresponds to the value received in the XLink Resolution Notification parameters.

`mpdURL` – If provided, this string shall provide the MPD URL to replace the Period containing the `xlink:href` attribute. One of either the `mpdURL` or `Period` properties shall be provided.

`period` – If provided, this string shall contain the Period text to replace the Period containing the `xlink:href` attribute. One of either the `mpdURL` or `period` properties shall be provided.

Response:

After receiving the XLink Resolved request, the Receiver may respond indicating that the default Period was successfully replaced with either the requested MPD URL or the supplied Period fragment. If the Receiver could not successfully complete the request, the disposition object shall contain a reason code with a description.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "xlink": {"type": "string", "format": "uri"},
    "disposition": {
      "type": "object",
      "properties": {
        "code": {"type": "integer"},
        "description": {"type": "string"}
      },
      "required": ["code"]
    },
    "timing": {
      "type": "object",
      "properties": {
        "currentPosition": {"type": "double"},
        "periodStart": {"type": "string", "format": "xs:dateTime"},
        "duration": {
          "type": "double",
          "minimum": 0.0
        }
      },
      "required": ["currentPosition", "periodStart", "duration"]
    }
  },
  "required": ["xlink", "disposition", "timing"]
}

```

xlink – This required string shall be the XLink value from the `xlink:href` attribute in the MPD **period** element. This string corresponds to the value received in the XLink Resolved Request parameters.

disposition – This required string shall provide the disposition of the replacement request. The **code** property shall indicate the result while the **description** property shall provide descriptive text of the disposition.

code – This required integer shall provide the disposition code of the replacement request.

- 0 – The requested replacement was successful
- 1 – The broadband MPD URL cannot be reached
- 2 – The local MPD URL cannot be found
- 3 – All locally cached segments are not present
- 4 – The request was not received in time to override the default content
- 5 – The provided XLink does not correspond to any known XLink
- 6 – The supplied **period** value was malformed and cannot be parsed.

description – This optional string shall provide descriptive text of the disposition of the replacement request. It is recommended that Broadcaster Applications only rely on the **code** parameter since **description** values are not standardized and may differ across implementations.

timing – This required object supplies detailed information regarding the playback of the `Period` containing the `XLink` intended to be replaced. If the replacement was successful, the timing information applies to the replacement content. Otherwise, the timing information applies to the default content.

currentPosition – This required double shall provide the time offset relative to the beginning of the `Period` containing the `XLink` in seconds. Note that the value may be negative if the response is received prior to the start of the replaced `Period`.

periodStart – This required `xs:dateTime` value shall provide the absolute time reference for the start (i.e., the zero time) of the media timeline of the `Period` containing the `XLink`. The value of `periodStart` represents the sum of the `Period` offset from the `MPD@availabilityStartTime` in the `MPD` containing the `Period` containing the `XLink`. The `xs:dateTime` XML data type shall be formatted as defined in the W3C XML Schema [28].

duration – This required double shall provide the total number of seconds defined by the `Period` containing the `XLink`.

As an example, the Broadcaster Application submits the following request:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.xlinkResolution",
  "params": {
    "xlink": "urn:xbc 4399FB77-3939EA47",
    "mpdURL": "localhost://xyz/ReplacementAd.mpd"
  },
  "id": 104
}
```

If the Receiver can replace the default content with the alternate `MPD` URL, it responds with the timing information associated with the replaced period and a disposition indicating successful replacement with the alternate period used as shown below.

```
<-- {
  "jsonrpc": "2.0"
  "result": {
    "xlink": "urn:xbc 4399FB77-3939EA47",
    "disposition": {
      "code": 0,
      "description": "Success"
    },
    "timing": {
      "currentPosition": -5.25,
      "periodStart": "2019-07-20T20:28:30.000Z",
      "duration": 30.0
    }
  },
  "id": 104
}
```

Alternatively, if the Receiver could not use the replacement `MPD` URL, as requested by the Broadcaster Application, it returns a disposition indicating the reason for the failed request. It additionally provides the timing information related to the default content.

```

<-- {
  "jsonrpc": "2.0",
  "result": {
    "xlink": "urn:xbc 4399FB77-3939EA47",
    "disposition": {
      "code": -4,
      "description": "Replacement request was too late"
    },
    "timing": {
      "currentPosition": 3.5,
      "periodStart": "2019-01-01T20:59:30.000Z",
      "duration": 30.0
    }
  },
  "id": 104
}

```

As a further example, the Broadcaster Application submits the following request containing the actual alternate period text instead of an mpdURL:

```

--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.xlinkResolution",
  "params": {
    "xlink": "urn:xbc 4399FB77-3939EA47",
    "period": "<Period start='PT0S'>
      <AdaptationSet mimeType='video/mp4' ... >
        <SegmentTemplate timescale='90000' ...
          media='alt-$Number$.mp4v' duration='90000'
          startNumber='32401' />
        <Representation id='v2' width='1920' height='1080' ... />
      </AdaptationSet>
    </Period>"
  },
  "id": 104
}

```

If the Receiver can replace the default content with the alternate period, it responds with the timing information associated with the replaced period and a disposition indicating successful replacement with the alternate period used as shown below.

```
<-- {
  "jsonrpc": "2.0"
  "result": {
    "xlink": "urn:xbc 4379FB99-3939AC47",
    "disposition": {
      "code": 0,
      "description": "Success"
    },
    "timing": {
      "currentPosition": -5.25,
      "periodStart": "2019-04-04T13:14:30.000Z",
      "duration": 30.0
    }
  },
  "id": 104
}
```

Annex A DASH Ad Insertion

A.1 INTRODUCTION

The Reference Receiver Model supports personalized client-side ad insertion. The mechanism depends on the type of service: for regular “watch TV” types of services that include application-based features, the Broadcaster Application can directly parse the MPD and resolve the XLink. It can also provide a service to the Receiver Media Player to resolve an XLink, but this assumes that the Receiver Media Player should parse the MPD and propagate the XLink resolution event to the Broadcaster Application in less than 150 milliseconds. Receiver Media Players that cannot parse the MPD and propagate the XLink resolution event in less than 150 milliseconds cannot be used for client-side ad insertion. For application-based services, the Broadcaster Application itself can determine the appropriate content to play for any particular user. The XLink-based method is described and specified here.

A.2 DYNAMIC AD INSERTION PRINCIPLES

The basic principles of dynamic ad insertion include:

- 1) Signaling an “ad avail” to indicate an available slot for a replacement ad or ad pod;
- 2) Controlling the pre-caching of ad replacement content;
- 3) Choosing the appropriate ad for download based on user preferences, cookies (past viewing, or past answers to questions), custom logic, etc.; and
- 4) Executing the seamless splice of replacement ad into broadcast stream

The method specified herein involves:

- 1) Signaling an “ad avail” as a DASH `Period` in an MPD;
- 2) Tagging each replaceable `Period` with an XLink;
- 3) Having the Receiver Media Player call on the Broadcaster Application to resolve each XLink;
- 4) Managing the pre-caching of ad content, if necessary, within the Broadcaster Application;
- 5) Employing the Broadcaster Application to choose the appropriate ad, personalized to this viewer; and
- 6) Having the Receiver Media Player execute the seamless splice;

Figure A.1 describes three DASH `Period`s, designated as P1, P2, and P3. Here, P2 is an “ad avail”, meaning that a possible replacement ad may be substituted for the content that would otherwise play during that interval. If no personalization is done, the default content (called “P2*”) plays. The asterisk indicates that the MPD element describing P2 includes an XLink. Based on some personalization criteria (described later), P2 could be replaced with an alternative content, shown as P2a, P2b, or P2c in the figure.

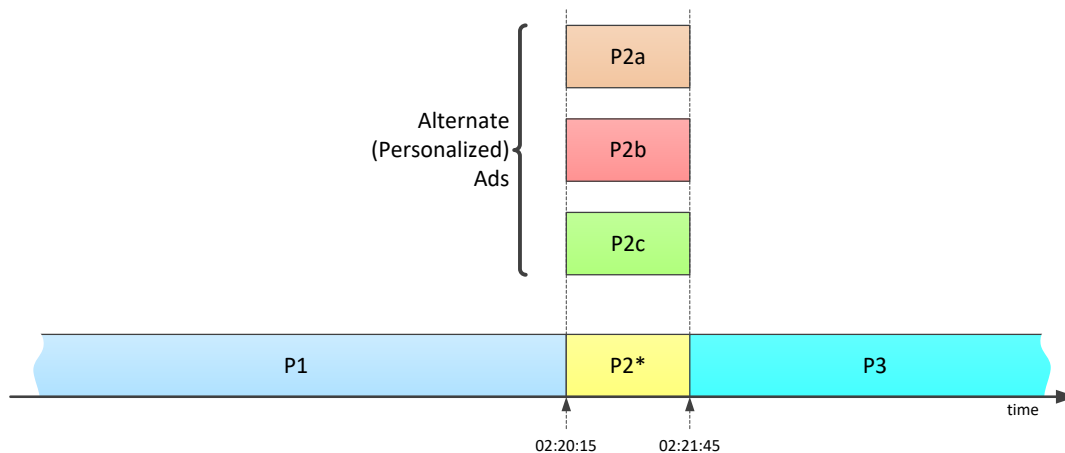


Figure A.1 Personalized ad periods.

In this example, the broadcast MPD includes an XLink attribute within the `period` element describing P2.

A.3 OVERVIEW OF XLINKS

The XLink concept is specified by W3C in [37]. While XLinks have many applications, the ATSC 3.0 runtime environment only uses one feature: an XLink appearing as an attribute of an XML element can be “resolved” to replace that entire element with the result of the resolution process. This feature is signaled by setting the XLink “show” attribute to the value “embed”, as shown in the simplified example MPD `Period` given in Figure A.2.

```
<Period start="PT9H" tag:atsc.org,2016:xlink:show="embed"
tag:atsc.org,2016:xlink:href="private-data-known-to-app"
duration="PT30S">
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>
```

Figure A.2 Example period with XLink.

When a Receiver Media Player that meets ATSC 3.0 performance requirements for XLink resolution is used to render media and it receives an updated MPD including an XLink in one or more `period` elements, if there is a Broadcaster Application currently running, it passes the contents of the `xlink:href` attribute as a parameter to the Broadcaster Application using the XLink Resolution API specified in Section 9.16 to attempt to get it resolved. If successful, the application responds with a replacement `Period` element. A broadcaster application that directly renders media will have access to the updated MPD and can resolve the XLink directly. For the example above, the replacement `Period` might look like the one given in Figure A.3.


```
<Period start="PT9H">
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="ad6-$Number$.mp4v" duration="90000" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>
```

Figure A.3 Example remote period.

In the example, the original content (referenced by the relative URL "xbc-\$Number\$.mp4v") was replaced by a personalized ad segment referenced by "ad6-\$Number\$.mp4v".

Annex B Obscuring the Location of Ad Avails

B.1 OBSCURING THE LOCATION OF AD AVAILS

The XLink-based technique described in Section A.3 may be used to obscure the location of ad avails, thanks to the possibility that the XLink resolution can return not one, but multiple period elements.

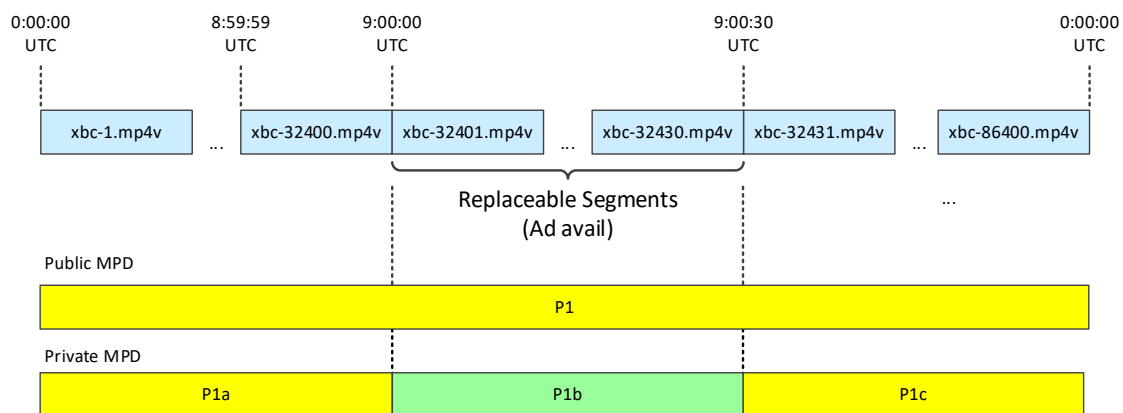


Figure B.1 Public and private MPDs.

Figure B.1 describes a “public” MPD and a “private” MPD. The public MPD is delivered in the broadcast emission stream, and signals one Period (P1) starting at midnight UTC on 1 July 2016 and lasting for 24 hours. Figure B.2 shows, in simplified form, this MPD instance.

```
<MPD type="dynamic" ... availabilityStartTime="2016-07-01T00:00:00Z">
...
<Period start="PT0S" duration="PT1D" tag:atsc.org,2016:xlink:show="embed"
  tag:atsc.org,2016:xlink:href="urn:xbc:d=2016-07-01&s=4A92D344092A9A09ec35" >
  <!-- P1 -->
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" duration="90000" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>
</MPD>
```

Figure B.2 Example public MPD.

The MPD includes an XLink within the one period. The resolution of this XLink yields three periods, replacing the one. This “private” MPD is shown in Figure B.1 at the bottom. Now, the

period P1 is broken into three sub-Periods, P1a, P1b, and P1c. The middle period, P1b, is actually an ad avail occurring at 9:00 am UTC and lasting for 30 seconds.

Note that the public MPD indicated Media Segments of 1-second duration running all day long. Thus, the first Media Segment of the day would be xbc-1.mp4v, running through to the end of the day with xbc-864000.mp4v.

Figure B.3 illustrates an MPD that would result in output of exactly the same media content as the MPD given in Figure B.2. This example is given only to illustrate the concept of the `Period.SegmentTemplate.startNumber`.

```
<MPD type="dynamic" ... availabilityStartTime="2016-07-01T00:00:00Z">
...
<Period start="PT0S" >    <!-- P1a -->
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" duration="90000" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>

<Period start="PT9H" >    <!-- P1b -->
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" duration="90000"
      startNumber="32401" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>

<Period start="PT9H0M30S" >    <!-- P1c -->
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" duration="90000"
      startNumber="32431" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>
</MPD>
```

Figure B.3 Example MPD equivalent.

Although there are two additional Periods, the media content referenced by the middle one (the Period starting at 9 a.m.) is exactly the same as before, because the starting index number is set to 32,401. This is the same number, and hence the same Media Segment, that would have played at 9 a.m. according to the MPD of Figure B.2. Likewise, at 30 seconds after 9:00, the start number of 32,431 references the same content as before.

Therefore, if XLink resolution would return these three Periods, nothing would change. A seamless ad replacement can occur, however, if the Broadcaster Application replaces the middle Period with different content. Figure B.4 illustrates the case that the content referenced in the middle Period is now to a personalized ad of 30 seconds duration, by reference to `media="ad7-$Number$.mp4v"`.

```

<MPD type="dynamic" ... availabilityStartTime="2016-07-01T00:00:00Z">
...
<Period start="PT0S" > <!-- P1a -->
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" duration="90000" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>

<Period start="PT9H" > <AdaptationSet mimeType="video/mp4" ... > <!-- P1b* -->
  <SegmentTemplate timescale="90000" ... media="ad7-$Number$.mp4v" duration="90000" />
  <Representation id="v2" width="1920" height="1080" ... />
</AdaptationSet>
</Period>

<Period start="PT9H0M30S" > <!-- P1c -->
  <AdaptationSet mimeType="video/mp4" ... >
    <SegmentTemplate timescale="90000" ... media="xbc-$Number$.mp4v" duration="90000"
      startNumber="32431" />
    <Representation id="v2" width="1920" height="1080" ... />
  </AdaptationSet>
</Period>
</MPD>

```

Figure B.4 Example ad replacement.

Annex C A/344:2019 WS API Changes

C.1 INTRODUCTION

This annex provides the detailed set of WebSocket API revisions made from the A/344:2017 publication of this standard to the A/344:2019 revision and is intended to preserve elements that are no longer normative requirements of the standard.

C.2 QUERY ALERTING URL AND ALERTING CHANGE NOTIFICATION APIS

A new property, `filteredEventList`, was added to both the Query Alerting URL and Alerting Change Notification APIs. The property lists the AEA events that have been filtered by the receiver. A detailed description is available in Sections 9.2.8 and 9.3.8, respectively.

In addition, the Alerting URL and Alerting Change Notification APIs have been changed to provide the alerting XML fragments instead of references to those fragments. The previous APIs have been deprecated in this revision and the previous text is provided in the following subsections.

C.2.1 Query Alerting URL API [Deprecated]

The Broadcaster Application may wish to access the various alerting metadata structures from the current broadcast. The Query Alerting URL API returns a list of URLs the Broadcaster Application can use to retrieve (for example, by XHR) the specific alerting metadata it has requested.

The Request Alerting URL API shall be defined as follows:

method: "org.atsc.query.alertingUrl"

params: A JSON object consisting of a key named `alertingTypes` with a list of enumerated values. An empty list is equivalent to supplying all values.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "alertingTypes": {
      "type": "array",
      "items": {
        "type": "string", "enum": ["AEAT", "OSN"]
      }
    }
  },
  "required": ["alertingTypes"]
}
```

`alertingTypes` – An array of one or both of the alerting types as follows:

AEAT – Requests a URL referencing the most recent AEAT.

OSN – Requests a URL referencing the most recent OSN.

Response:

result: a JSON object containing a list of references as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "urlList": { "type": "array", "items": { "type": "object",
      "properties": {
        "alertingType": { "type": "string", "enum": ["AEAT", "OSN"] },
        "alertingUrl": { "type": "string", "format": "url" },
        "receiveTime": { "type": "string", "format": "xs:dateTime" },
        "filteredEventList": {
          "type": "array",
          "items": {
            "aeaId": { "type": "string" }
          }
        }
      }
    }
  },
  "required": ["alertingType", "alertingUrl"]
},
{
  "required": ["urlList"]
}
```

alertingType – One of the alerting types. The corresponding **alertingUrl** can be used to access the data corresponding to the type of alerting metadata fragment indicated.

alertingUrl – A fully-qualified URL that can be used by the Broadcaster Application, for example in an XHR request, to retrieve the current broadcast alerting metadata for the associated **alertingType**. Alerting is delivered in XML fragments whose syntax is defined in A/331 [1].

receiveTime – The date and time the alerting fragment was received. This value shall be provided when the object is “OSN”. (Note: The **OnscreenMessageNotification** element includes a **keepScreenClear@notificationDuration** attribute which is the duration of the KeepScreenClear message starting from the time the OSN was received. Thus, the time the OSN was received is necessary for the Broadcaster Application to fully utilize the OSN information.) The **xs:dateTime** XML data type shall be formatted as defined in the W3C XML Schema [28].

filteredEventList – Provides a list of AEA events that have been filtered by the Receiver. The Receiver may elect to filter an event for a variety of reasons based on user preferences, location or some other criteria. If an AEA event is filtered, the corresponding **AEAT.AEA@aeaId** shall appear in the **filteredEventList** property. If an AEA event is not filtered, the corresponding **AEAT.AEA@aeaId** shall not appear in the list. An empty or absent **filteredEventList** indicates that no events have been filtered by the Receiver. This property is only applicable when the **alertingType** is “AEAT”.

aeaId – The **AEAT.AEA@aeaId** of an AEA event that has been filtered by the Receiver. See the previous description of the **filteredEventList** property for the usage of this property.

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.alertingUrl",
  "params": {
    "alertingTypes": ["AEAT", "OSN"]
  },
  "id": 913
}
```

The Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {
    "urlList": [
      { "alertingType": "AEAT",
        "alertingUrl": "http://127.0.0.1:8080/.lls/AEAT.xml" },
      { "alertingType": "OSN",
        "alertingUrl": "http://127.0.0.1:8080/.lls/OSN.xml",
        "receiveTime": "2017-01-01T23:54:59.590Z" }
    ]
  },
  "id": 913
}
```

It should be noted that the `AEAT.xml` and `OSN.xml` files referenced contain the AEAT and OSN XML fragments as described in A/331 [1], respectively. The Receiver shall extract the XML fragment from the binary LLS table and GZIP encoding.

C.2.2 Alerting Change Notification API [Deprecated]

The Alerting Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if there is a change to the version of the AEAT or OSN alerting data structure and the Broadcaster Application has subscribed to receive such notifications via the API specified in Section 9.7.5. Note that the receipt of a new alerting object without a previous receipt is considered a version change. The Alerting Change Notification may also be issued if alerting event filtering has been changed resulting in a change to the filtered events.

The notification message contains a list of URLs referencing the new or updated alerting fragments.

The Alerting Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` key with a value "alertingChange" and a key named `urlList` which is an array of items containing an `alertingType` as defined in the enumerated list, a URL to the metadata fragments and a `receiveTime` if the URL points to an OSN fragment.

params JSON Schema:

```

{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["alertingChange"]},
    "urlList": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "alertingType": {"type": "string", "enum": ["AEAT", "OSN"]},
          "alertingUrl": {"type": "string", "format": "uri"},
          "receiveTime": {"type": "string", "format": "xs:dateTime"},
          "filteredEventList": {
            "type": "array",
            "items": {
              "aeald": {"type": "string"}
            }
          }
        }
      }
    },
    "required": ["alertingType", "alertingUrl"]
  },
  "required": ["msgType", "urlList"]
}

```

alertingType – One of "AEAT" or "OSN", required. The corresponding **alertingUrl** can be used to access the XML fragment corresponding to the **alertingType**.

alertingUrl – A required fully-qualified URL that can be used by the Broadcaster Application, for example in an XHR request, to retrieve the current alerting XML fragment for the associated **alertingType**.

receiveTime – The date/time the OSN fragment was received, optional. This value shall be presented when the **alertingType** is "OSN". (Note: The OSN table includes a Notification Duration field which is the duration of the KeepScreenClear message starting from the time the OSN was received. Thus, the time the OSN was received is necessary for the Broadcaster Application to fully utilize the OSN information.) The **xs:dateTime** XML data type shall be formatted as defined in the W3C XML Schema [28].

filteredEventList – Provides a list of AEA events that have been filtered by the Receiver. The Receiver may elect to filter an event for a variety of reasons based on user preferences, location or some other criteria. If an AEA event is filtered, the corresponding **AEAT.AEA@aeaId** shall appear in the **filteredEventList** property. If an AEA event is not filtered, the corresponding **AEAT.AEA@aeaId** shall not appear in the list. An empty or absent **filteredEventList** indicates that no events have been filtered by the Receiver. This property is only applicable when the **alertingType** is "AEAT".

Note that if the filtering criteria change due to, for example, the Receiver moving, or the user changing preferences, previously filtered events may become unfiltered and events that were previously unfiltered may now be filtered. In this case, the Alerting Change Notification shall

be issued by the Receiver with a new `filteredEventList`. The Broadcaster Application may need to take alternative actions based on the new list of events.

`aeaId` – The `AEAT.AEA@aeaId` of an AEA event that has been filtered by the Receiver. See the previous description of the `filteredEventList` property for the usage of this property.

For example, the Receiver may indicate that a new AEAT has been received by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "alertingChange",
    "urlList": [
      { "alertingType": "AEAT",
        "alertingUrl": " http://127.0.0.1:8080/.lls/aeat.xml" }
    ]
  }
}
```

As a further example, the Receiver may indicate that a new AEAT and OSN have been received by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "alertingChange",
    "urlList": [
      { "alertingType": "AEAT",
        "alertingUrl": "http://127.0.0.1:8080/.lls/AEAT.xml" },
      { "alertingType": "OSN",
        "alertingUrl": "http://127.0.0.1:8080/.lls/OSN.xml",
        "receiveTime": "2017-01-01T23:54:59.590Z" }
    ]
  }
}
```

Note that the `AEAT.xml` file referenced contains the AEAT XML fragment extracted from the LLS table as described in A/331 [1] while `OSN.xml` contains the OSN XML fragment defined in the same referenced document. Further, the prefix shown in these examples are informative only. The Receiver implementation may provide access to the fragments through the Application Context Cache hierarchy or completely outside of it. The Broadcaster Application should make no assumptions regarding the path and simply use it to access the fragment data directly.

C.3 EXTENDED VIDEO SCALING AND POSITIONING API

The previous revision contained only integer values which limited the position and size that could be defined for the RMP video display. The values were all changed from integer to double and the appropriate values were placed in the examples. Due to the lack of strong typing in JavaScript, providing integer inputs to this API will result in the same behavior as before. With the current API definition in Section 9.7.2, the full resolution of the display may now be used.

C.4 EXTENDED SET RMP URL API

In the previous revision, the Set RMP URL request contained a single, normative property, `rmpurl`, which specified that the RMP should begin playing the specified MPD. The previous API is provided below:

method: "org.atsc.setRMPURL"

params: A JSON object consisting of a key named `rmpurl`.

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "rmpurl": {"type": "string"},
  },
  "required": ["rmpurl"]
}
```

`rmpurl` – This required string shall be a URL referencing an MPD to be played by the RMP. The URL shall be accessible to the Receiver.

The `rmpurl` property remains as defined previously but now is governed by an additional operation property as described in Section 9.7.3.

C.5 ADDED APIS

The Integrated Subscribe API (Section 9.7.5.1) and the Integrated Unsubscribe API (Section 9.7.5.2) were added to the set of WebSocket APIs in this revision. These two APIs replace the other subscription APIs present in the initial release of the standard.

C.6 DEPRECATED APIS

The following APIs have been deprecated as of this revision. These APIs are copies of the text that was present in the previous revision. Please refer to Table 9.1 for information on applicability and replacement APIs.

C.6.1 XLink Resolution API [Deprecated]

An XLink Resolution request shall be issued by the Receiver to the currently executing Broadcaster Application when in the MPD the Receiver Media Player (RMP) encounters a **Period**@`xlink:href` attribute in the form of a tag URI beginning with `tag:atsc.org,2016:xlink`. Any XLinks not beginning with this URI are expected to be disregarded or processed in a proprietary manner. For XLinks beginning with `tag:atsc.org,2016:xlink`, the Receiver shall request that the Broadcaster Application resolve the XLink, using the XLink Resolution API to return one or more **Period** elements to be used by the RMP to replace the **Period** element in which the XLink appeared. The Receiver may not let the RMP replace the Period in the MPD with the resolved period elements if it recognizes that it cannot replace all segments in the Period. Especially, if the resolved period elements in the response specify segment files in the Application Context Cache but they are not present in the Application Context Cache, then the Receiver shall not let the RMP replace the Period in the MPD.

The XLink Resolution API shall be defined as follows:

method: "org.atsc.xlinkResolution"

params: A JSON object consisting of a key named `xlink` and a string representing the contents of the `"xlink:href"` element.

params JSON Schema:

```
{
  "type": "object",
  "properties": {"xlink": {"type": "string"}},
  "required": ["xlink"]
}
```

xlink – This required string shall be the XLink value from the `xlink:href` attribute in the MPD **Period** element.

Response:

result: A JSON object containing a "resolution" key whose string value represents one or more **Period** elements. The `BaseURL` of the **Period** elements shall be considered to be the root of the Application Context Cache. If a segment file specified in the **Period** elements has a relative URL, then Receiver shall use the segments file with the relative URL under the Application Context Cache. If a segment file specified in the **Period** elements has an absolute URL, it shall be the URL of the broadband server from which the segment file may be retrieved. The URL includes the "https" protocol identifier.

result JSON Schema:

```
{
  "type": "object",
  "properties": {"resolution": {"type": "string"}},
  "required": ["resolution"]
}
```

For example, the Receiver notifies the Broadcaster Application of an XLink:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.xlinkResolution",
  "params": {"xlink": "urn:xbc 4399FB77-3939EA47"},
  "id": 5
}
```

Upon success, the Broadcaster Application might respond:

```
--> {
  "jsonrpc": "2.0",
  "result": {"resolution": "<Period start='PT9H'> <AdaptationSet
    mimeType='video/mp4' /> <SegmentTemplate timescale='90000'
    media='xbc-$Number$.mp4v' duration='90000' startNumber='32401' />
    <Representation id='v2' width='1920' height='1080' />"},
  "id": 5
}
```

Note the use of single quotes for all attributes in the Period (required for proper JSON value syntax).

If the Broadcaster Application is unable to resolve the XLink, it can respond with an error code -9:

```
--> {
  "jsonrpc": "2.0",
  "error": {"code": -9, "message": "XLink cannot be resolved"},
  "id": 59
}
```

C.6.2 Subscribe MPD Changes API [Deprecated]

The Subscribe MPD Changes API can be used by a Broadcaster Application to be notified whenever the version of the broadcast MPD currently in use by the RMP changes. Once subscribed, the Receiver notifies the Broadcaster Application when any version change occurs by issuing the MPD Change Notification API specified in Section 9.3.8. Notifications continue until an Unsubscribe MPD Changes API is issued, or until the Service is changed.

The Subscribe MPD Changes API shall be defined as follows:

method: "org.atsc.subscribeMPDChange"
 params: none.

For example, the Broadcaster Application can subscribe to MPD changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.subscribeMPDChange",
  "id": 55
}
```

Upon success, the Broadcaster Application would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 55
}
```

C.6.3 Unsubscribe MPD Changes API [Deprecated]

The Unsubscribe MPD Changes API can be issued by a Broadcaster Application to stop receiving notifications of MPD changes.

The Unsubscribe MPD Changes API shall be defined as follows:

method: "org.atsc.unsubscribeMPDChange"
 params: none.

For example, the Broadcaster Application can subscribe to MPD changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.unsubscribeMPDChange",
  "id": 56
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 56
}
```

C.6.4 Subscribe Alerting Changes API [Deprecated]

The Subscribe Alerting Changes API can be used by a Broadcaster Application to be notified whenever new or new versions of alerting metadata, specifically AEAT and OSN LLS fragments, are received. Once subscribed, the Receiver shall notify the Broadcaster Application when any addition or version change occurs by issuing the Alerting Change Notification API specified in Section 9.3.8. A notification is also issued if an AEAT or OSN table is currently available avoiding the need for the Broadcaster Application to issue a query (Section 9.2.8) immediately after subscribing to notifications. Notifications shall continue until an Unsubscribe Alerting Changes API is issued, or until the Broadcaster Application is no longer active.

The Subscribe Alerting Changes API shall be defined as follows:

method: "org.atsc.subscribeAlertingChange"
 params: A JSON object consisting of a key named `alertingTypes` with a list of enumerated values shown. An empty list is equivalent to supplying all values.
 params JSON Schema:

```
{
  "type": "object",
  "properties": {"alertingTypes": {
    "type": "array",
    "items": {"type": "string", "enum": ["AEAT", "OSN"]}
  }},
  "required": ["alertingTypes"]
}
```

`alertingTypes` — An array of alerting object names as follows:

AEAT — Requests that the Broadcaster Application be notified if a new or updated AEAT is received. A notification shall be immediately issued if an AEAT is currently available. If

an executing Broadcaster Application has been suspended while a receiver-native user interface is being presented, the notification should be issued as soon as the Broadcaster Application is re-launched, if the alert is still active.

OSN – Requests that the Broadcaster Application be notified if a new or updated OSN is received. A notification shall be immediately issued if an OSN is currently active.

For example, the Broadcaster Application can subscribe to alerting changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.subscribeAlertingChange",
  "params": {
    "alertingTypes": ["AEAT", "OSN"]
  },
  "id": 55
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 55
}
```

The Receiver would then notify the Broadcaster Application whenever new or updated AEAT or OSN was received or if either fragment is currently present.

C.6.5 Unsubscribe Alerting Changes API [Deprecated]

The Unsubscribe Alerting Changes API can be issued by a Broadcaster Application to stop receiving notifications of alerting updates and additions.

The Unsubscribe Alerting Changes API shall be defined as follows:

method: "org.atsc.unsubscribeAlertingChange"

params: A JSON object consisting of a key named `alertingTypes` with a list of enumerated values shown. An empty list shall indicate that the Broadcaster Application is unsubscribing from all current alerting notification subscriptions.

params JSON Schema:

```
{
  "type": "object",
  "properties": {"alertingTypes": {
    "type": "array",
    "items": {"type": "string", "enum": ["AEAT", "OSN"]}
  },
  "required": ["alertingTypes"]
}}
```

`alertingTypes` – An array of alerting metadata types. An empty array shall indicate all alerting metadata types.

For example, the Broadcaster Application can unsubscribe from AEAT alerting changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.unsubscribeAlertingChange",
  "params": {
    "alertingTypes": ["AEAT"]
  },
  "id": 56
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 56
}
```

In this example, assuming the Broadcaster Application had subscribed to both alerting types, then the unsubscribe operation only applies to the AEAT notifications. Notifications of the OSN receipt will continue.

To unsubscribe from all alerting notifications, the Broadcaster Application can issue the following request:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.unsubscribeAlertingChange",
  "params": {
    "alertingTypes": []
  },
  "id": 312
}
```

C.6.6 Subscribe Content Changes API [Deprecated]

The Subscribe Content Changes API can be used by a Broadcaster Application to be notified whenever a new package or new version of a package is received. Once subscribed, the Receiver notifies the Broadcaster Application when a new or updated package has been received by issuing the Content Change Notification API specified in Section 9.3.9. Notifications continue until an Unsubscribe Content Changes API is issued, or until the Broadcaster Application is no longer active.

The Subscribe Content Changes API shall be defined as follows:

method: "org.atsc.subscribeContentChange"

params: none.

For example, the Broadcaster Application can subscribe to content changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.subscribeContentChange",
  "id": 58
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 58
}
```

The Receiver would then notify the Broadcaster Application whenever new packages are received resulting in changes to files in the Application Context Cache. Note that this notification only occurs after the files within the listed packages have been made available to the Broadcaster Application through the Receiver Web Server having matched the Application Context ID and any defined Filter Codes. In other words, the Broadcaster Application can immediately start using the files associated with the notified packages.

C.6.7 Unsubscribe Content Changes API [Deprecated]

The Unsubscribe Content Changes API can be issued by a Broadcaster Application to stop receiving notifications of content updates and additions.

The Unsubscribe Content Changes API shall be defined as follows:

method: "org.atsc.unsubscribeContentChange"

params: none.

For example, the Broadcaster Application can unsubscribe from content changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.unsubscribeContentChange",
  "id": 72
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 72
}
```

C.6.8 Subscribe RMP Media Time Change Notification API [Deprecated]

The Subscribe RMP Media Time Change Notification API can be used by a Broadcaster Application to be notified whenever the current playback position of the content being presented by the RMP changed. Once subscribed, the Receiver notifies the Broadcaster Application whenever the current playback position of the content being presented by the RMP changed by

issuing the RMP Media Time Change Notification API specified in Section 9.14.5. Notifications continue until an Unsubscribe RMP Media Time Change Notification API is issued, or until the Broadcaster Application is no longer active.

The Subscribe RMP Media Time Change Notification API shall be defined as follows:

method: "org.atsc.subscribeRmpMediaTimeChange"

params: none.

For example, the Broadcaster Application can subscribe to RMP media time changes by issuing:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.subscribeRmpMediaTimeChange",  
  "id": 358  
}
```

Upon success, the Receiver would respond:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {},  
  "id": 358  
}
```

The Receiver would then notify the Broadcaster Application whenever the current playback position of the content being presented by the RMP changed.

C.6.9 Unsubscribe RMP Media Time Change Notification API [Deprecated]

The Unsubscribe RMP Media Time Change Notification API can be issued by a Broadcaster Application to stop receiving notifications of changes to the current playback position of content being presented by the RMP.

The Unsubscribe RMP Media Time Change Notification API shall be defined as follows:

method: "org.atsc.unsubscribeRmpMediaTimeChange"

params: none.

For example, the Broadcaster Application can unsubscribe from notifications of changes to the RMP media time by issuing:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.unsubscribeRmpMediaTimeChange",  
  "id": 372  
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 372
}
```

C.6.10 Subscribe RMP Playback State Change Notification API [Deprecated]

The Subscribe RMP Playback State Change Notification API can be used by a Broadcaster Application to be notified whenever the current playback state of the content being presented by the RMP changed. Once subscribed, the Receiver notifies the Broadcaster Application whenever the current playback state of the content being presented by the RMP changed by issuing the RMP Playback State Change Notification API specified in Section 9.14.6. Notifications continue until an Unsubscribe RMP Playback State Change Notification API is issued, or until the Broadcaster Application is no longer active.

The Subscribe RMP Playback State Change Notification API shall be defined as follows:

method: "org.atsc.subscribeRmpPlaybackStateChange"
params: none.

For example, the Broadcaster Application can subscribe to RMP playback state changes by issuing:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.subscribeRmpPlaybackStateChange",
  "id": 359
}
```

Upon success, the Receiver would respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {},
  "id": 359
}
```

The Receiver would then notify the Broadcaster Application whenever the playback state of the content being presented by the RMP changed.

C.6.11 Unsubscribe RMP Playback State Change Notification API [Deprecated]

The Unsubscribe RMP Playback State Change Notification API can be issued by a Broadcaster Application to stop receiving notifications of changes to the current playback state of content being presented by the RMP.

The Unsubscribe RMP Playback State Change Notification API shall be defined as follows:

method: "org.atsc.unsubscribeRmpPlaybackStateChange"
params: none.

For example, the Broadcaster Application can unsubscribe from notifications of changes to the RMP playback state by issuing:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.unsubscribeRmpPlaybackStateChange",  
  "id": 373  
}
```

Upon success, the Receiver would respond:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {},  
  "id": 373  
}
```

C.6.12 Subscribe RMP Playback Rate Change Notification API [Deprecated]

The Subscribe RMP Playback Rate Change Notification API can be used by a Broadcaster Application to be notified whenever the current playback rate of the content being presented by the RMP changed. Once subscribed, the Receiver notifies the Broadcaster Application whenever the current playback rate of the content being presented by the RMP changed by issuing the RMP Playback Rate Change Notification API specified in Section 9.14.7. Notifications continue until an Unsubscribe RMP Playback Rate Change Notification API is issued, or until the Broadcaster Application is no longer active.

The Subscribe RMP Playback Rate Change Notification API shall be defined as follows:

method: "org.atsc.subscribeRmpPlaybackRateChange"

params: none.

For example, the Broadcaster Application can subscribe to RMP playback rate changes by issuing:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.subscribeRmpPlaybackRateChange",  
  "id": 379  
}
```

Upon success, the Receiver would respond:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {},  
  "id": 379  
}
```

The Receiver would then notify the Broadcaster Application whenever the playback Rate of the content being presented by the RMP changed.

C.6.13 Unsubscribe RMP Playback Rate Change Notification API [Deprecated]

The Unsubscribe RMP Playback Rate Change Notification API can be issued by a Broadcaster Application to stop receiving notifications of changes to the current playback rate of content being presented by the RMP.

The Unsubscribe RMP Playback Rate Change Notification API shall be defined as follows:

method: "org.atsc.unsubscribeRmpPlaybackRateChange"

params: none.

For example, the Broadcaster Application can unsubscribe from notifications of changes to the RMP playback rate by issuing:

```
--> {  
  "jsonrpc": "2.0",  
  "method": "org.atsc.unsubscribeRmpPlaybackRateChange",  
  "id": 373  
}
```

Upon success, the Receiver would respond:

```
<-- {  
  "jsonrpc": "2.0",  
  "result": {},  
  "id": 373  
}
```

Annex D A/344:2020 WS API Changes

D.1 INTRODUCTION

This annex provides the detailed set of WebSocket API revisions made from the A/344:2019 publication of this standard to the A/344:2020 revision and is intended to preserve elements that are no longer normative requirements of the standard.

D.2 ADDED APIS

A general API to allow requests to be canceled was added in this revision (see Section 8.3.1 Cancel Request Command). Other additions include the Service Guide URLs APIs (Sections 9.2.9 and 9.3.10) and the Signaling Data APIs (Sections 9.2.10 and 9.3.11).

D.3 DEPRECATED APIS

The following APIs have been deprecated as of this revision. These APIs are copies of the text that was present in the previous revision. Please refer to Table 9.1 for information on applicability and replacement APIs.

D.3.1 Query MPD URL API [Deprecated]

The Broadcaster Application may wish to access the current broadcast DASH MPD. The Query MPD URL API returns a URL the Broadcaster Application can use to retrieve (for example, by XHR) the MPD.

The Request MPD URL API shall be defined as follows:

method: "org.atsc.query.MPDUrl"

params: Omitted

Response:

result: a JSON object containing an object as defined below.

result JSON Schema:

```
{
  "type": "object",
  "properties": {
    "MPDUrl": {"type": "string", "format": "uri"}
  },
  "required": ["MPDUrl"]
}
```

MPDUrl – A fully-qualified URL that can be used by the Broadcaster Application, for example in an XHR request, to retrieve the current broadcast MPD.

For example, the Broadcaster Application makes a query:

```
--> {
  "jsonrpc": "2.0",
  "method": "org.atsc.query.MPDUrl",
  "id": 913
}
```

The Receiver might respond:

```
<-- {
  "jsonrpc": "2.0",
  "result": {"MPDUrl": "http://127.0.0.1:8080/10.4/MPD.mpd"},
  "id": 913
}
```

D.3.2 MPD Change Notification API [Deprecated]

The MPD Change Notification API shall be issued by the Receiver to the currently executing Broadcaster Application if there is a change to the currently playing MPD (including a change to a different MPD or a new version of the currently playing MPD) and the Broadcaster Application has subscribed to receive such notifications via the API specified in Section 9.7.5. The Broadcaster Application may respond to the notification of a change to the MPD by using the MPD URL discovered using the Query MPD URL API specified in Section 9.2.7 to fetch a new copy.

The MPD Change Notification API shall be defined as follows:

method: "org.atsc.notify"

params: A JSON object consisting of a key named `msgType` with value "MPDChange".

params JSON Schema:

```
{
  "type": "object",
  "properties": {
    "msgType": {"type": "string", "enum": ["MPDChange"]},
  },
  "required": ["msgType"]
}
```

For example, the Receiver may indicate that a new MPD is available by issuing this JSON-RPC command:

```
<-- {
  "jsonrpc": "2.0",
  "method": "org.atsc.notify",
  "params": {
    "msgType": "MPDChange"
  }
}
```

Annex E JSON-RPC 2.0 Specification

The contents of this Annex were copied verbatim from <http://www.jsonrpc.org/specification> [34] on September 28, 2017.

Origin Date:

2010-03-26 (based on the 2009-05-24 version)

Updated:

2013-01-04

Author:

[JSON-RPC Working Group](#) <json-rpc@googlegroups.com>

1 Overview

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments. It uses [JSON \(RFC 4627\)](#) as data format.

It is designed to be simple!

2 Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#).

Since JSON-RPC utilizes JSON, it has the same type system (see <http://www.json.org> or [RFC 4627](#)). JSON can represent four primitive types (Strings, Numbers, Booleans, and Null) and two structured types (Objects and Arrays). The term "Primitive" in this specification references any of those four primitive JSON types. The term "Structured" references either of the structured JSON types. Whenever this document refers to any JSON type, the first letter is always capitalized: Object, Array, String, Number, Boolean, Null. True and False are also capitalized.

All member names exchanged between the Client and the Server that are considered for matching of any kind should be considered to be case-sensitive. The terms function, method, and procedure can be assumed to be interchangeable.

The Client is defined as the origin of Request objects and the handler of Response objects. The Server is defined as the origin of Response objects and the handler of Request objects.

One implementation of this specification could easily fill both of those roles, even at the same time, to other different clients or the same client. This specification does not address that layer of complexity.

3 Compatibility

JSON-RPC 2.0 Request objects and Response objects may not work with existing JSON-RPC 1.0 clients or servers. However, it is easy to distinguish between the two versions as 2.0 always has a member named "jsonrpc" with a String value of "2.0" whereas 1.0 does not. Most 2.0 implementations should consider trying to handle 1.0 objects, even if not the peer-to-peer and class hinting aspects of 1.0.

4 Request object

A rpc call is represented by sending a Request object to a Server. The Request object has the following members:

jsonrpc

A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".

method

A String containing the name of the method to be invoked. Method names that begin with the word rpc followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else.

params

A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id

An identifier established by the Client that MUST contain a String, Number, or NULL value if included. If it is not included it is assumed to be a notification. The value SHOULD normally not be Null¹ and Numbers SHOULD NOT contain fractional parts²

The Server MUST reply with the same value in the Response object if included. This member is used to correlate the context between the two objects.

4.1 Notification

A Notification is a Request object without an "id" member. A Request object that is a Notification signifies the Client's lack of interest in the corresponding Response object, and as such no Response object needs to be returned to the client. The Server MUST NOT reply to a Notification, including those that are within a batch request.

Notifications are not confirmable by definition, since they do not have a Response object to be returned. As such, the Client would not be aware of any errors (like e.g. "Invalid params", "Internal error").

4.2 Parameter Structures

¹ The use of Null as a value for the id member in a Request object is discouraged, because this specification uses a value of Null for Responses with an unknown id. Also, because JSON-RPC 1.0 uses an id value of Null for Notifications this could cause confusion in handling.

² Fractional parts may be problematic, since many decimal fractions cannot be represented exactly as binary fractions.

If present, parameters for the rpc call **MUST** be provided as a Structured value. Either by-position through an Array or by-name through an Object.

- by-position: params **MUST** be an Array, containing the values in the Server expected order.
- by-name: params **MUST** be an Object, with member names that match the Server expected parameter names. The absence of expected names **MAY** result in an error being generated. The names **MUST** match exactly, including case, to the method's expected parameters.

5 Response object

When a rpc call is made, the Server **MUST** reply with a Response, except for in the case of Notifications. The Response is expressed as a single JSON Object, with the following members:

jsonrpc

A String specifying the version of the JSON-RPC protocol. **MUST** be exactly "2.0".

result

This member is **REQUIRED** on success. This member **MUST NOT** exist if there was an error invoking the method. The value of this member is determined by the method invoked on the Server.

error

This member is **REQUIRED** on error. This member **MUST NOT** exist if there was no error triggered during invocation. The value for this member **MUST** be an Object as defined in section I.5.1.

id

This member is **REQUIRED**. It **MUST** be the same as the value of the id member in the Request Object. If there was an error in detecting the id in the Request object (e.g. Parse error/Invalid Request), **MUST** be Null. Either the result member or error member **MUST** be included, but both members **MUST NOT** be included.

5.1 Error object

When a rpc call encounters an error, the Response Object **MUST** contain the error member with a value that is a Object with the following members:

code

A Number that indicates the error type that occurred. This **MUST** be an integer.

message

A String providing a short description of the error. The message **SHOULD** be limited to a concise single sentence.

data

A Primitive or Structured value that contains additional information about the error. This may be omitted. The value of this member is defined by the Server (e.g. detailed error information, nested errors etc.).

The error codes from and including -32768 to -32000 are reserved for pre-defined errors. Any code within this range, but not defined explicitly below is reserved for future use. The error codes

are nearly the same as those suggested for XML-RPC at the following url: http://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php

code	message	meaning
-32700	Parse error	Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.
-32600	Invalid Request	The JSON sent is not a valid Request object.
-32601	Method not found	The method does not exist / is not available.
-32602	Invalid params	Invalid method parameter(s).
-32603	Internal error	Internal JSON-RPC error.
-32000 to -32099	Server error	Reserved for implementation-defined server-errors.

The remainder of the space is available for application defined errors.

6 Batch

To send several Request objects at the same time, the Client MAY send an Array filled with Request objects.

The Server should respond with an Array containing the corresponding Response objects, after all of the batch Request objects have been processed. A Response object SHOULD exist for each Request object, except that there SHOULD NOT be any Response objects for notifications. The Server MAY process a batch rpc call as a set of concurrent tasks, processing them in any order and with any width of parallelism.

The Response objects being returned from a batch call MAY be returned in any order within the Array. The Client SHOULD match contexts between the set of Request objects and the resulting set of Response objects based on the id member within each Object.

If the batch rpc call itself fails to be recognized as a valid JSON or as an Array with at least one value, the response from the Server MUST be a single Response object. If there are no Response objects contained within the Response array as it is to be sent to the client, the server MUST NOT return an empty Array and should return nothing at all.

7 Examples

Syntax:

```
--> data sent to Server
<-- data sent to Client
```

rpc call with positional parameters:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
```

```
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}
```

rpc call with named parameters:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
```

```
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}
```

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42, "subtrahend": 23}, "id": 4}
```

```
<-- {"jsonrpc": "2.0", "result": 19, "id": 4}
```

a Notification:

```
--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
```

```
--> {"jsonrpc": "2.0", "method": "foobar"}
```

rpc call of non-existent method:

```
--> {"jsonrpc": "2.0", "method": "foobar", "id": "1"}
```

```
<-- {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "1"}
```

rpc call with invalid JSON:

```
--> {"jsonrpc": "2.0", "method": "foobar", "params": "bar", "baz]}
```

```
<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}
```

rpc call with invalid Request object:

```
--> {"jsonrpc": "2.0", "method": 1, "params": "bar"}
```

```
<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}
```

rpc call Batch, invalid JSON:

```
--> [  
  {"jsonrpc": "2.0", "method": "sum", "params": [1,2,4], "id": "1"},  
  {"jsonrpc": "2.0", "method"
```

```
]  
  
<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}
```

rpc call with an empty Array:

```
--> []
```

```
<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}
```

rpc call with an invalid Batch (but not empty):

```
--> [1]
```

```
<-- [  

```

```
{ "jsonrpc": "2.0", "error": { "code": -32600, "message": "Invalid Request" }, "id":
null}
]
```

rpc call with invalid Batch:

```
--> [1,2,3]
<-- [
  { "jsonrpc": "2.0", "error": { "code": -32600, "message": "Invalid Request" }, "id":
null},
  { "jsonrpc": "2.0", "error": { "code": -32600, "message": "Invalid Request" }, "id":
null},
  { "jsonrpc": "2.0", "error": { "code": -32600, "message": "Invalid Request" }, "id":
null}
]
```

rpc call Batch:

```
--> [
  { "jsonrpc": "2.0", "method": "sum", "params": [1,2,4], "id": "1"},
  { "jsonrpc": "2.0", "method": "notify_hello", "params": [7]},
  { "jsonrpc": "2.0", "method": "subtract", "params": [42,23], "id": "2"},
  { "foo": "boo"},
  { "jsonrpc": "2.0", "method": "foo.get", "params": { "name": "myself" }, "id":
"5"},
  { "jsonrpc": "2.0", "method": "get_data", "id": "9" }
]
<-- [
  { "jsonrpc": "2.0", "result": 7, "id": "1"},
  { "jsonrpc": "2.0", "result": 19, "id": "2"},
  { "jsonrpc": "2.0", "error": { "code": -32600, "message": "Invalid Request" },
"id": null},
  { "jsonrpc": "2.0", "error": { "code": -32601, "message": "Method not found" },
"id": "5"},
  { "jsonrpc": "2.0", "result": ["hello", 5], "id": "9" }
]
```

rpc call Batch (all notifications):

```
--> [
  { "jsonrpc": "2.0", "method": "notify_sum", "params": [1,2,4]},
  { "jsonrpc": "2.0", "method": "notify_hello", "params": [7]}
]
<-- //Nothing is returned for all notification batches
```

8 Extensions

Method names that begin with `rpc.` are reserved for system extensions, and **MUST NOT** be used for anything else. Each system extension is defined in a related specification. All system extensions are **OPTIONAL**.

Copyright (C) 2007-2010 by the JSON-RPC Working Group

This document and translations of it may be used to implement JSON-RPC, it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way.

The limited permissions granted above are perpetual and will not be revoked.

This document and the information contained herein is provided "AS IS" and ALL WARRANTIES, EXPRESS OR IMPLIED are DISCLAIMED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

— End of Document —