INTERNATIONAL STANDARD

ISO/IEC 23009-6

First edition
2017-12

Information technology — Dynamic adaptive streaming over HTTP (DASH) —

Part 6: DASH with server push and WebSockets

Technologies de l'information — Diffusion adaptative dynamique sur HTTP (DASH) —

Partie 6: DASH avec serveur de poussée et protocoles WebSocket

Reference number
ISO/IEC 23009-6:2017(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information.*

A list of all parts in the ISO/IEC 23009 series can be found on the ISO website.

Introduction

Dynamic Adaptive Streaming over HTTP (DASH) is intended to support a media-streaming model for delivery of media content in which the control lies exclusively with the client.

This document specifies carriage of MPEG DASH media presentations over full duplex HTTP-compatible protocols, particularly HTTP/2 (version 2 of the HTTP protocol as defined by the IETF in Reference [8]) and WebSocket (WebSocket protocol as defined by the IETF in RFC 6455). This carriage takes advantage of the capabilities of these protocols to optimize delivery of MPEG DASH media presentations.

Information technology — Dynamic adaptive streaming over HTTP (DASH) —

Part 6:
DASH with server push and WebSockets

1 Scope

This document specifies carriage of MPEG-DASH media presentations over full duplex HTTP-compatible protocols, particularly HTTP/2 and WebSocket. This carriage takes advantage of the features these protocols support over HTTP/1.1 to improve delivery performance, while still maintaining backwards compatibility, particularly for the delivery of low latency live video.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEEE 1003.1-2008, *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX), Base Specifications, Issue 7*

IETF RFC 3986, *Uniform Resource Identifiers (URI): Generic Syntax, January 2005*

IETF RFC 6455, *The WebSocket Protocol, December 2011*

IETF RFC 7158, *The JavaScript Object Notation (JSON) Data Interchange Format, March 2013*

IETF RFC 7231, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, June 2014*

3 Terms, definitions, abbreviated terms and conventions

3.1 Terms and definitions

For the purposes of this document, the following terms, definitions, abbreviated terms and conventions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

3.1.1
push acknowledgement
Push Ack
response modifier, sent from a server to a client, which enables a server to state the *push strategy* (3.1.3) used when processing a request

3.1.2
push directive
request modifier, sent from a client to a server, which enables a client to express its expectations regarding the server's *push strategy* (3.1.3) for processing a request

3.1.3
push strategy
segment transmission strategy, that defines the ways in which segments may be pushed from a server to a client

3.1.4
DASH server push
push
transmission of a segment from server to client based on a *push strategy* (3.1.3), as opposed to directly in response to a client request

3.2 Conventions

NOTE In this document, data formats are described using the ABNF method as described in RFC 5234.

STRING = 1* VCHAR

INTEGER = 1* DIGIT

PPCHAR= `%x21 / %x23-7E`

SQUOTE= `%x27`

UCHAR= `%x21 / %x23-7A / %x7C / %x7E`

4 Background

The basic mechanisms of MPEG-DASH over HTTP/1.1 can be augmented by utilizing the new features and capabilities that are provided by the more recent Internet protocols such as HTTP/2 and WebSocket; see Annex A for several illustrative use cases. While HTTP/2 and WebSocket are quite different in details, they both allow server-initiated and client-initiated transactions, data request cancelation and multiplexing of multiple data responses.

While in the case of HTTP/2 it is possible to carry DASH presentations without additional support, these new capabilities can be used to reduce the transmission delay (latency). Also, both HTTP/2 and WebSocket are designed to interoperate with existing HTTP/1.1 infrastructure, allowing for graceful fallback to HTTP/1.1 when the more recent protocol is not available.

The overall workflow of MPEG-DASH over these protocols is shown in Figure 1. The client and server first initiate a media channel where the server can actively push data to the other (enabled by HTTP/2 server push or WebSocket messaging). The media channel may be established via the HTTP/1.1 protocol upgrade mechanism or by some other means. After the connection is established, the DASH client requests the media or the MPD from the server, with a URI and a push strategy. This strategy informs the server about how the client would like media delivery to occur (initiated by the server or initiated by the client). Once the server receives the request, it responds with the requested data and initializes the push cycle as defined in the push strategy. Annex B shows a typical end-to-end video streaming system over HTTP/2 that can benefit from signalling and messages defined in this document.

Figure 1 shows an example DASH session wherein the client requests the MPD first and then the media segments with a push strategy. Initialization data are pushed in response to a push strategy associated to the MPD request. After receiving the requested MPD, the client starts requesting video segments from the server with the respective DASH segment URL and a segment push strategy. Then, the server responds with the requested video segment, followed by the push cycles as indicated by the segment push strategy. Typically, the client starts playing back the video after a minimum amount of data is received and then the aforementioned process repeats until the end of the media streaming session.

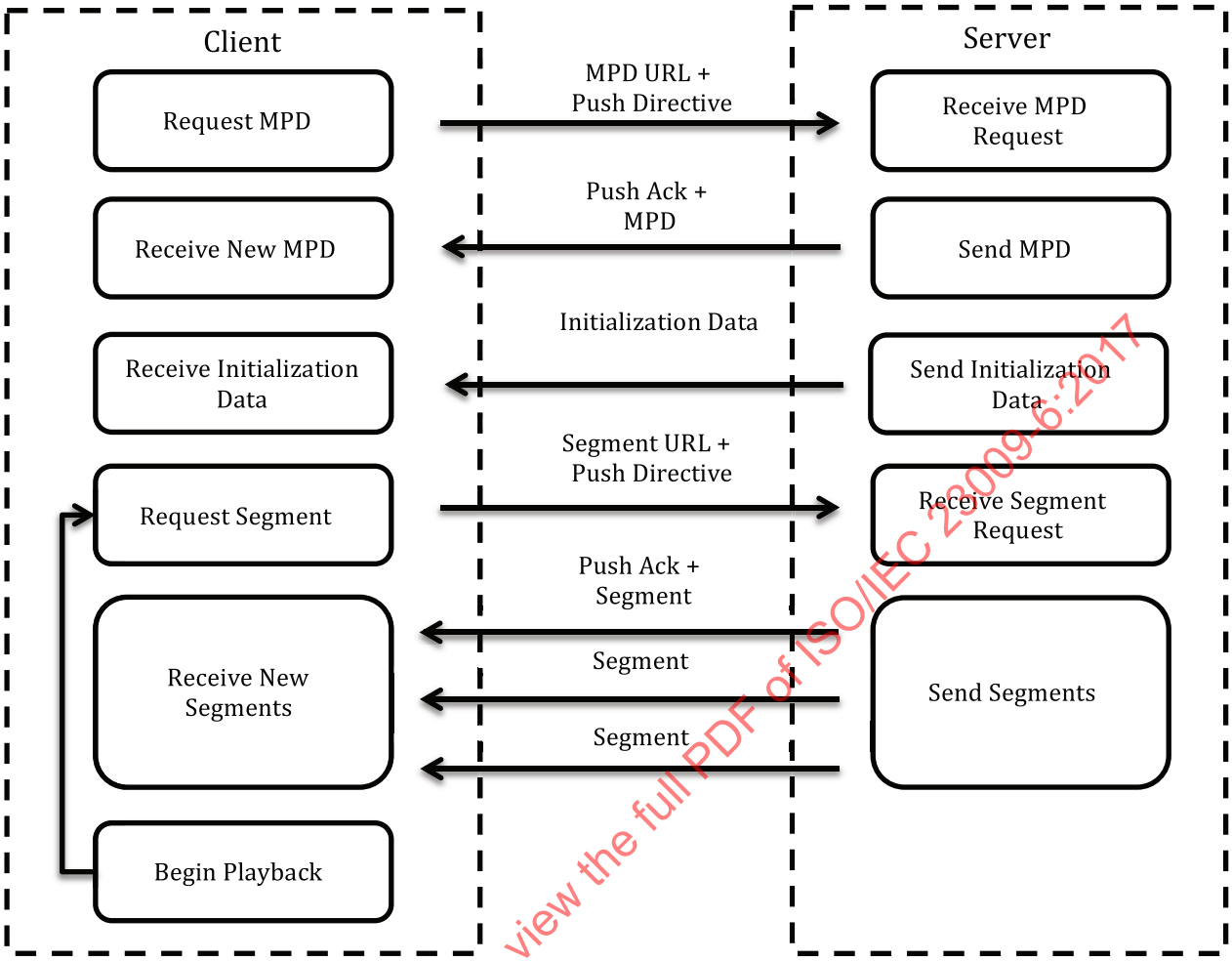


Figure 1 — Overall flow of video streaming using DASH server push

5 Specification structure

This document defines the signalling and message formats for driving the delivery of MPEG-DASH media presentations over full-duplex HTTP-compatible protocols. Details are provided for utilizing this signalling over the HTTP/2 (Clause 7) and the WebSocket (Clause 8) protocols.

Annex C provides examples of HTTP/2 client/server behaviour implementing signalling and message formats defined in this document. Annex D provides examples of WebSocket client/server behaviour implementing signalling and message formats defined in this document. Annex E illustrates the HTTP protocol upgrade and fallback procedure for WebSocket. These informational annexes are provided to demonstrate the use of the specified signalling and message formats to build streaming systems that take advantage of the full-duplex capabilities of the underlying transport protocol.

6 Definitions

6.1 Data type definitions

6.1.1 General

Clause 6 describes a number of primitive data types (see Table 1) used to define the signalling over protocols addressed in this document. Details for implementing these primitives for a given protocol may be found in the subclause of this document defining that binding.

Table 1 — Definitions of primitive data types

Data type	Base type	Description
`BinaryObject`	N/A	An untyped binary object made up of 0 or more bytes.
`Boolean`	N/A	A true or false value.
`MPD`	MPD	An MPEG-DASH Media Presentation Description (MPD), as defined in ISO/IEC 23009-1.
`Null`	N/A	An empty value.
`PushAck`	String	A response from the server acknowledging a push request. The PushAck contains the accepted values for the push strategy specified in the PushDirective. For details, see 6.1.4.
`PushDirective`	String	A directive describing the requested push strategy to be employed within the streaming session. For details, see 6.1.3.
`Segment`	Segment	An MPEG-DASH initialization or media segment, as defined in ISO/IEC 23009-1.
`String`	N/A	A UTF-8 character string.
`URI`	String	A Uniform Resource Identifier (URI), as defined in RFC 3986.
`URLList`	String	A list of URLs. For details, see 6.1.5.
`URLTemplate`	String	A URL template and corresponding parameters that describe a set of URLs. For details, see 6.1.6.

6.1.2 PushType

A `PushType` is the description of a push strategy. It contains a name identifying the push strategy and possibly its associated parameters.

The format of a `PushType` in the ABNF form is as follows:

PUSH_TYPE = PUSH_TYPE_NAME [OWS ";" OWS PUSH_PARAMS]

PUSH_TYPE_NAME = DQUOTE <URN> DQUOTE

PUSH_PARAMS = PUSH_PARAM *(OWS ";" OWS PUSH_PARAM)

PUSH_PARAM = 1*PPCHAR

Where,

'<URN>' syntax is defined in RFC 2141. Valid values for this URN according to this document are defined in Table 3,

'OWS' is defined in RFC 7230, 3.2.3 and represents optional whitespace.

The definition of PUSH_PARAMS is generic to allow the definition of new push strategies without any limitation on their parameters. Each push strategy adds some restriction on the number and on the definitions of the PUSH_PARAM instances used with it. Valid values for PUSH_PARAMS are defined in Table 4.

EXAMPLE If the push strategy expects a parameter of type 'INTEGER', then there is only one 'PUSH_PARAM' defined by 'INTEGER' as in 3.2. If it expects a parameter of type 'URLTemplate', then there is only one 'PUSH_PARAM' defined by 'URLTemplate' as in 6.1.6.

6.1.3 PushDirective

A `PushDirective` signals the push strategy that a client would like the server to use for delivery of one or more future segments. A `PushDirective` has a type (described in Table 3) and depending on the type, may have one or more additional parameters associated with it (described in Table 4).

In general, a client may signal one or more PushDirectives for a single message. The server may select at most one of the provided push strategies. This mechanism allows for clients to interoperate with servers that allow different push strategies and for forward compatibility, as the new types of push strategies are introduced.

The format of a `PushDirective` in the ABNF form is as follows:

PUSH_DIRECTIVE = PUSH_TYPE [OWS ";" OWS QVALUE]

PUSH_TYPE = <A PushType defined in 6.1.2>

QVALUE = <a qvalue, as defined in RFC 7231>

When multiple push directives are applied to a request, a client may apply a quality value ("qvalue") as is described for use in content negotiation in RFC 7231. A client may apply higher quality values to directives it wishes to take precedence over alternative directives with a lower quality value. Note that these values are hints to the server and do not imply that the server will necessarily choose the strategy with the highest quality value. If the quality value "qvalue" is not present, the default quality value is 1,0.

6.1.4 PushAck

A Push Acknowledgement (`PushAck`) is sent from the server to the client to indicate that the server intends to follow a given push strategy. At most, one Push Acknowledgment may be returned, indicating the push strategy that is in effect at the server. A Push Acknowledgment, depending on the type, may have one or more additional parameters associated with it (described in Table 4).

The format of the `PushAck` in the ABNF form is as follows:

PUSH_ACK = PUSH_TYPE

Where PUSH_TYPE is defined in 6.1.2.

6.1.5 URLList

A `URLList` describes a specific set of URLs as a delimited list. A client may use a list to explicitly signal the segments to be pushed during a push transaction. The list of URLs describes the sequence of segments to be pushed within this push transaction.

The `URLList` string format ABNF follows:

URL_LIST = LIST_ITEM *(OWS ";" OWS LIST_ITEM)

LIST_ITEM = 1*PPCHAR

Each list element is formed as a URL as defined in RFC 3986. If the URL is in relative form, it is considered relative to the segment being requested. See Annex F for examples of the URL list under various scenarios.

6.1.6 URLTemplate

A `URLTemplate` describes a specific set of URLs via a template and the corresponding parameters required to expand the template. A client may use a template to explicitly signal the segments to be pushed during a push transaction. The string is formed as a list of individual URL templates, each of

which may be parameterized to signal one or more URL values. When fully evaluated, the complete list of URLs describes the sequence of segments to be pushed within this push transaction.

The `URLTemplate` format is inspired by the "level 1" URI template scheme defined in IETF RFC 6570.

NOTE The above template mechanism may be used to describe URLs contained in the MPEG-DASH MPD, whether they are formed using a SegmentTemplate or SegmentList. It is not possible to use `URLTemplate` to describe URLs formed via SegmentTemplate when they use `$Time$` variable, unless the time value of each segment can be predicted or is described via SegmentTimeline, typically when @r is present and is not negative.

In addition, each parameter may be suffixed with an additional format tag aligned with the printf() format tag as defined in IEEE 1003.1-2008 following this prototype:

%0[width]d

The width parameter is an unsigned integer that provides the minimum number of characters to be printed. If the value to be printed is shorter than this number, the result shall be padded with zeros. The value is not truncated even if the result is larger.

The `URLTemplate` string format ABNF follows:

URL_TEMPLATE = TEMPLATE_ITEM *(OWS ";" OWS TEMPLATE_ITEM)

TEMPLATE_ITEM = SQUOTE TEMPLATE_ELEMENT SQUOTE [OWS ":" OWS "{" OWS TEMPLATE_PARAMS OWS "}"]

TEMPLATE_ELEMENT = CLAUSE_LITERAL [CLAUSE_VAR [CLAUSE_LITERAL]]

CLAUSE_LITERAL = 1*UCHAR

CLAUSE_VAR = "{%0" 1*DIGIT "d}" / "{}"

TEMPLATE_PARAMS = VALUE_LIST / VALUE_RANGE

VALUE_LIST = 1*DIGIT *(OWS "," OWS 1*DIGIT)

VALUE_RANGE = 1*DIGIT OWS "-" OWS 1*DIGIT

Each template element is formed as a URL as defined in RFC 3986, containing up to one macro for parameterization. If the URL is in relative form, it is considered relative to the segment being requested.

The {} parameter is used to specify a specified list or range of URLs that differ by segment number or timestamp and is expanded using the provided value specifier. If no parameter is provided, the value specifier is optional. This makes it possible to provide a simple list of URLs.

The URL list will be generated from each template item by evaluating the provided parameter. For number ranges, this means generating a URL for each segment number in the range provided (inclusive).

The complete URL list is formed by expanding each URL template in turn, creating an ordered list of URLs. See Annex F for examples of the push template under various scenarios.

6.1.7 FastStartParams

A fast start parameter set (`FastStartParams`) is sent from the client to the server to signal the client's preferences for initialization information and media, which may be used by a server to determine the most appropriate set of segments to push to the client in response to an MPD request.

The parameter set is expressed as a set attributes, made up of keys or key/value pairs. Each attribute shall be treated as AND conditions.

The `FastStartParams` string format ABNF is as follows:

FAST_START_PARAMS = ATTRIBUTE_LIST / ATTRIBUTE_ITEM

ATTRIBUTE_LIST = ATTRIBUTE_ITEM OWS ";" OWS ATTRIBUTE_LIST / ATTRIBUTE_ITEM

ATTRIBUTE_ITEM = 1*PPCHAR

With OWS (optional whitespace) as defined in IETF RFC 7230 3.2.3

Table 2 describes valid values for PushStartParams attributes:

Table 2 — Valid attributes for FastStartParams

Attribute Type	ABNF	Description
Initialization segments only	`ATTRIBUTE_ITEM = "init-only"`	Only initialization segments should be pushed. If not present, both Initialization Segment and some Media Segments may be pushed.

Media Type	`ATTRIBUTE_ITEM =`
	`"type=" MEDIATYPE`
	`MEDIATYPE = "video" / "audio~""`
		Only Initialization and/or Media Segments related to MEDIATYPE should be pushed.

Start bitrate	`ATTRIBUTE_ITEM = "bitrate=" RATE` `RATE = SQUOTE INTEGER SQUOTE`	Only Initialization and/or Media Segments from the Representation whose bitrate (in bit per second) is the nearest but not greater than the specified value should be pushed. Note Selection of the Representation from which segments are pushed may be determined by @bandwidth attribute for Representations.
Resolution	`ATTRIBUTE_ITEM = "height=" RESOLUTION` `RESOLUTION = SQUOTE INTEGER SQUOTE`	When provided with "video" media type attribute, only Initialization and/or Media Segments from the Representation whose number of horizontal lines is the nearest and preferably not greater than specified value.
Language	`ATTRIBUTE_ITEM = "lang=" SQUOTE STRING SQUOTE`	Declares the language code for segments to be pushed. The syntax and semantics according to IETF RFC 5646 shall be used.

Media amount	`ATTRIBUTE_ITEM =`
	`"D=" SQUOTE INTEGER SQUOTE /`
	`"B=" SQUOTE INTEGER SQUOTE`
		Declares a maximum amount of Media data to be pushed.
		This limit can be expressed as a maximum duration `"D"` in milliseconds or a number of bytes `"B"`.

Media starting point	`ATTRIBUTE_ITEM = "t=" START_POINT` `START_POINT = "begin" / "now"`	Declares the desired media starting point of Initialization Segments and/or Media Segments to be pushed.
URL List	`ATTRIBUTE_ITEM = "urls=[" URL_LIST "]"`	Describes the list of segment URLs that may be returned by the server, where URL_LIST is as defined in 6.1.5. Shall only be included in a Push Acknowledgment.

Each attribute is optional. If the attribute list is empty, it shall be interpreted as including "init-only". See Annex G for examples of fast start under various scenarios.

6.2 Push strategy definitions

Table 3 provides the description of each PUSH_TYPE_NAME defined in this document.

Table 3 — Valid values for PUSH_TYPE_NAME

PushType	Description
urn:mpeg:dash:serverpush:2017:push-fast-start	Indication that, along with an MPD, initialization data and optionally a given number of initial media segments are considered for push. A server receiving such push strategy may push some or all available Initialization Segments and optionally some media segments related to the requested MPD within the constraints defined by provided attributes[a]. A client receiving such push strategy is informed that a server intends to push some or all available Initialization Segments and optionally some Media Segments within the constraints defined by provided attributes [a]. If attributes are not specified, the server may push what it considers the most appropriate[a] by default.
urn:mpeg:dash:serverpush:2017:push-list	Indication that some segments as described by the URL list are considered for push. A server receiving such push strategy may use it to identify some segments to push. A client receiving such push strategy can be informed on the segments the server intends to push.
urn:mpeg:dash:serverpush:2017:push-next	Indication that the next K segments in the order of time, using the requested segment as the initial index, are considered for push. A server receiving such push strategy may push the next segments consecutively to the requested one. A client receiving such push directive is informed that the server intends to push the next segments consecutively to the requested one.
urn:mpeg:dash:serverpush:2017:push-none	Indication that no push should occur. A server receiving such push strategy should prevent from pushing. A client receiving such push directive is informed that the server does not intend to push.
urn:mpeg:dash:serverpush:2017:push-template	Indication that some segments as described by the URL template are considered for push. A server receiving such push strategy may use it to identify some segments to push. A client receiving such push directive can be informed on the segments the server intends to push.
urn:mpeg:dash:serverpush:2017:push-time	Indication that the next segments in the order of time, continuing until the segment time (presentation time of the first frame) of a segment exceeds time, T, are considered for push. A server receiving such push strategy may push a given duration of media segments. A client receiving such push directive is informed that the server intends to push a given duration of media segments.
[a] To identify more focused resources to push at the beginning to achieve a fast start, the DASH server may use, in addition to specified attributes, client hints, client preferences, client logs, MPD knowledge or its own proprietary knowledge of how the segments are generated.	

A server shall recognize at least the "urn:mpeg:dash:serverpush:2017:push-none" strategy.

Each push strategy may only be valid when applied to a segment request, an MPD request or both. Table 4 describes the type of request for which each strategy may be applied and describes the parameters that may be used.

Table 4 — Valid request types and parameters for each PushType

PushType	Request type	PUSH_PARAM
urn:mpeg:dash:serverpush:2017:push-fast-start	MPD	FastStartParams
urn:mpeg:dash:serverpush:2017:push-list	Segment	URLList
urn:mpeg:dash:serverpush:2017:push-next	Segment	INTEGER
urn:mpeg:dash:serverpush:2017:push-none	MPD or Segment	N/A
urn:mpeg:dash:serverpush:2017:push-template	Segment	URLTemplate
urn:mpeg:dash:serverpush:2017:push-time	Segment	INTEGER

7 DASH server push over HTTP/2

7.1 PushDirective binding

In HTTP/2, push directives are signalled using an HTTP header called “Accept-Push-Policy”. The content of this header is a PushDirective as specified in 6.1.3.

NOTE As required by HTTP, multiple push directives can be signalled either using multiple HTTP headers or by combining multiple push directives as a comma-separated list into a single header. Annex H illustrates the use of push directives with the switching element from ISO/IEC 23009-1.

7.2 PushAck binding

In HTTP/2, push acknowledgements are signalled using an HTTP header called “Push-Policy”. The content of this header is a PushAck as specified in 6.1.4.

Additionally, the server may advertise the PushType(s) it supports by using an HTTP header called “Supported-Push-Policies“. The content of this header is a comma-separated list of PUSH_TYPE_NAME(s) as specified in 6.1.2.

NOTE Supported-Push-Policies can be used by the server when it is unable to support any of the requested PushTypes.

7.3 Push cancel

In HTTP/2, a client may explicitly request to cancel ongoing push requests using an HTTP header called “Push-Cancel“. The content of this header is a URLList as defined in 6.1.5.

A client sending a “Push-Cancel” with a URLList parameter informs the server that on-going and promised pushes for resources with one of the specified URLs can be cancelled for the current connection. If no outstanding push requests are in effect, then this header will have no effect.

The server may send a RST_STREAM frame (see IETF RFC 7540, 6.4) with a cancel code on streams corresponding to the on-going and promised pushes for resources with one of the specified URLs.

NOTE Server behaviour is best effort and optional.

8 DASH server push over WebSocket

8.1 Message flow over WebSocket

Figure 2 shows the message flow for carrying an MPEG-DASH media presentation over a full duplex WebSocket session. Messages are defined to allow for MPD and segment objects to be delivered over a WebSocket sub-protocol (see Table 5). These messages may carry push directives that signal additional segment objects to be delivered over the WebSocket channel. Note that this flow is identical to the general message flow described in Clause 4, using WebSocket-specific message bindings.

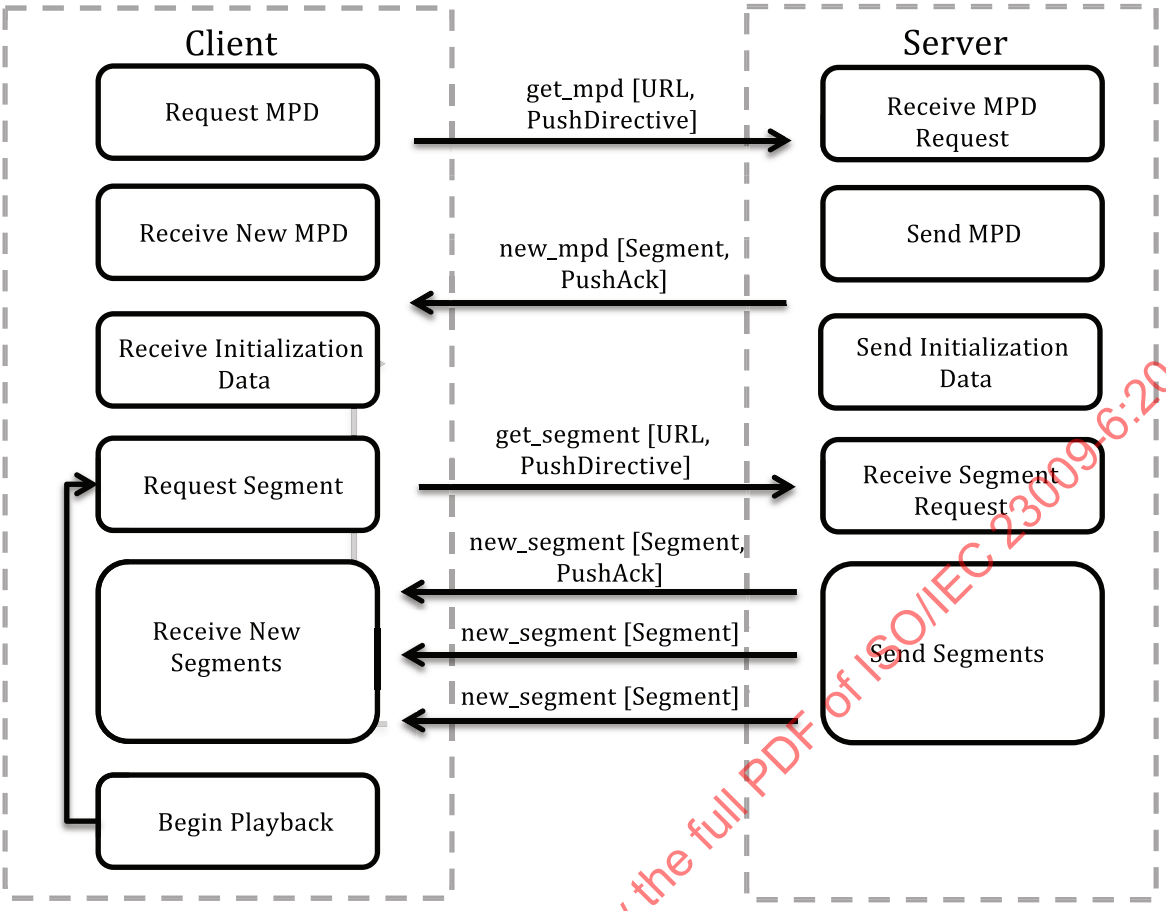


Figure 2 — Message flow over WebSocket

8.2 WebSocket sub-protocol for MPEG-DASH

8.2.1 MPEG-DASH WebSocket frame format and semantics

The DASH sub-protocol shall use the "binary" format (opcode "binary" or any '"continuation" frames thereof) for all messages exchanged over the WebSocket connection, as described in RFC 6455.

The MPEG-DASH sub-protocol frame consists of a frame header and frame payload. The frame header (see Figure 3) shall be formed as WebSocket frame Extension Data, which shall be present and of which the size can be determined as 4+4*EXT_LENGTH bytes as given by the DASH sub-protocol frame header. The frame payload corresponds to the WebSocket Application data, as described in RFC 6455.

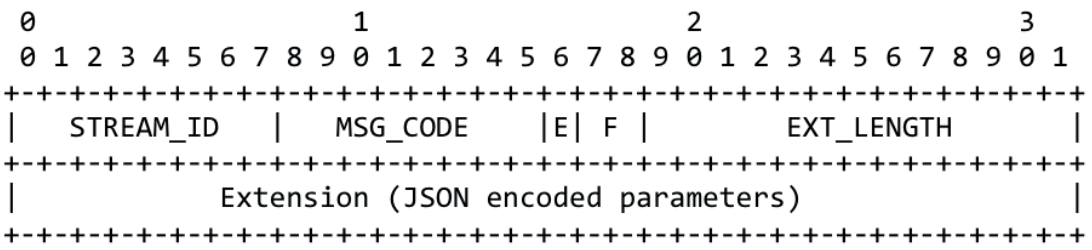


Figure 3 — DASH sub-protocol frame header for WebSocket

The DASH sub-protocol frame header is defined as follows:

STREAM_ID: 8 bits

Is an identifier of the current stream, which allows multiplexing of multiple requests/responses over the same WebSocket connection. The responses to a particular request shall use the same STREAM_ID as that request. The appearance of a new STREAM_ID indicates that a new stream has been initiated. The reception of a cancel request, an end of stream or an error shall result in closing the stream identified by the carried STREAM_ID.

MSG_CODE: 8 bits

Indicates the MPEG-DASH message represented by this frame. Available message codes are defined in 8.3.

E: 1 bit

This field is the error flag. When this field is set, the receiver may interpret the message as an error. Additional information about the error may be available in the extension header.

F: 2 bits

Reserved.

EXT_LENGTH: 13 bits

Provides the length in 4 bytes of the extension data that precedes the application data, including padding.

Extension: 4*EXT_LENGTH

The extension header shall be a JSON encoding of additional information fields that apply to the request/response, conforming to RFC 7158. To align with 4 byte boundaries, padding 0 bytes may be added after the extension header. The content shall be encoded in UTF-8 format. The JSON encoding of the extension header shall consist of a single root-level JSON object, containing zero or more name/value pairs.

8.2.2 Definition of WebSocket streams

The DASH sub-protocol for WebSocket defines the concept of streams that allows for an independent, bi-directional, sequence of frames to be exchanged between the client and server. Multiple streams may be created on top of the same WebSocket connection. The server shall send responses to the client's requests on the same stream that was used to submit the request. For instance, a push response that contains a set of segments shall use the same stream for the delivery of all resources of the response.

This is different from the behaviour in HTTP/2, where each resource will be assigned to a separate stream. The streams are identified by their STREAM_ID as defined in 8.2.1.

Each stream shall only carry at most one push directive and its responses. New push directives shall be started in a new stream.

8.3 WebSocket message codes

Table 5 — List of available DASH sub-protocol message codes

Message code	Message	Definition
1	get_mpd	8.4.1
2	get_segment	8.4.2
3	new_mpd	8.4.3
4	new_segment	8.4.4
5	end_of_stream	8.4.5
255	segment_cancel	8.4.6

8.4 WebSocket message definitions

8.4.1 MPD request (client → server)

The MPD request message initiates the request for a DASH MPD file. One or more push directives may be provided with the MPD request.

An optional headers field may be included to provide additional information to the server to aid in processing the message.

— Message name: get_mpd

— Supplied arguments

Parameter name	Type	Cardinality	Description
mpd_uri	URI	1	The full URI for the MPD being requested.
push_directive	PushDirective	0..N	A push strategy to be applied to this MPD request, as described in 6.1.3.
headers	String	0..1	Contains a CRLF separated set of HTTP 1.1 conformant header fields that apply to this message.

The supplied arguments shown above shall be JSON encoded conforming to normative JSON schema shown.

```
{
  "$schema":"http://json–schema.org/draft–04/schema#",
  "title": "Get MPD",
  "description": "Normative schema for get_mpd message .",
  "type": "object",
  "properties": {
    "mpd_uri": {
      "type": "string",
      "format": "uri"
    },
    "push_directive": {
      "type": "array",
      "items": {"type": "string"},
      "minItems": 0
    },
    "headers": {
      "type": "string"
    }
  },
  "required": ["mpd_uri"]
}
```

8.4.2 Segment request (client → server)

The segment request message initiates the request for a DASH segment. The segment request may include one or more push directives to inform the server to actively push one or more future segments.

An optional headers field may be included to provide additional information to the server to aide in processing the message.

— Message name: get_segment

— Supplied arguments

Parameter name	Type	Cardinality	Description
segment_uri	URI	1	The full URI for the video segment being requested.
push_directive	PushDirective	0..N	The desired push strategy for getting following segments, as described in 6.1.3.
headers	String	0..1	Contains a CRLF separated set of HTTP 1.1 conformant header fields that apply to this message.

The supplied arguments shown above shall be JSON encoded conforming to normative JSON schema shown below.

```
{
  "$schema":"http://json–schema.org/draft–04/schema#",
  "title": "Segment Request",
  "description": "Normative schema for get_segment message .",
  "type": "object",
  "properties": {
    "segment_uri": {
      "type": "string",
      "format": "uri"
    },
    "push_directive": {
      "type": "array",
      "items": {"type": "string"},
      "minItems": 0
    },
    "headers": {
      "type": "string"
    }
  },
  "required": ["segment_uri"]
}
```

8.4.3 MPD received (server → client)

This message represents the server's response from a previous *get_mpd* message sent by the client.

The presence of at most one push acknowledgment informs the client on the push strategy to be taken by the server in response to a push directive, including possibly that no push strategy will be in effect. A push acknowledgment shall only be applied to the first response of a push sequence and not to following pushed responses.

An optional headers field may be included to provide additional information to the client to aide in processing the message. A status code is included to signal additional detail about the contents of the message.

— Message name: new_mpd

— Supplied arguments

Parameter name	Type	Cardinality	Description
mpd	String	1	The MPD returned by the server.
push_ack	PushAck	0..1	The push strategy that the server will follow, as described in 6.1.4.
headers	String	0..1	Contains a CRLF separated set of HTTP 1.1 conformant header fields that apply to this message.
status	integer	1	An HTTP status code, conforming to RFC 7231, Clause 6, which applies to this message.

The supplied arguments shown above shall be JSON encoded conforming to normative JSON schema shown below.

```
{
   "$schema":"http://json–schema.org/draft–04/schema#",
   "title": "MPD Received",
   "description": "Normative schema for new_mpd message .",
   "type": "object",
   "properties": {
      "mpd": {
         "type": "string"
      },
      "push_ack": {
         "type": "string"
      },
      "headers": {
         "type": "string"
      },
      "status": {
         "type": "integer"
      }
   },
   "required": ["mpd","status"]
}
```

For the parameter "mpd", JSON data type of "string" is used.

8.4.4 Segment received (server → client)

This message represents the server's response from a previous *get_segment* message sent by the client. A server may issue multiple responses for a single request, as appropriate for the push strategy in the corresponding *get_segment* message.

The presence of at most one push acknowledgment informs the client on the push strategy to be taken by the server in response to a push directive, including possibly that no push strategy will be in effect. A push acknowledgment shall only be applied to the first response of a push sequence and not to following pushed responses.

An optional headers field may be included to provide additional information to the client to aide in processing the message. A status code is included to signal additional detail about the contents of the message.

— Message name: new_segment

— Supplied arguments

Parameter name	Type	Cardinality	Description
`Segment_URL`	string	1	The segment URL.
`push_ack`	PushAck	0..1	The push strategy that the server will follow, as described in 6.1.4.
`headers`	String	0..1	Contains a CRLF separated set of HTTP 1.1 conformant header fields that apply to this message.
`status`	integer	1	An HTTP status code, conforming to RFC 7231 Clause 6, which applies to this message.

The supplied arguments shown above shall be JSON encoded conforming to normative JSON schema shown below.

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title": "Segment Received",
  "description": "Normative schema for new_segment message .",
  "type": "object",
  "properties": {
    "segment_URL": {
      "type": "string"
    },
    "push_ack": {
      "type": "string"
    },
    "headers": {
      "type": "string"
    },
    "status": {
```



```
        ""type": "integer"
      }
    },
    "required": ["segment_URL","status"]
}
```

After the JSON encoded parameters which shall only include "`segment_URL`", "`push_ack`", "`headers`", "`status`", the segment data shall be sent as binary data without JSON encoding. No other JSON encoded parameters shall follow the segment data.

8.4.5 End of stream (EOS) (server → client)

This message is sent by the server to indicate that a previous operation cannot be continued as a result of a change to resource availability or other condition. An example of such situation is when a Period of unknown duration comes to an end and the server is not able to push segments of that Representation anymore. The EOS message shall result in the closing of the stream.

An optional headers field may be included to provide additional information to the client to aide in processing the message. If no "`headers`" parameter is included in the end_of_stream message, then the EXT_LENGTH shall be set to 0 and no empty JSON parameter encoding shall be present after EXT_LENGTH field.

— Message name: end_of_stream

— Supplied arguments

Parameter name	Type	Cardinality	Description
`headers`	String	0..1	Contains a CRLF separated set of HTTP 1.1 conformant header fields that apply to the request corresponding to this stream.

The supplied arguments shown above shall be JSON encoded conforming to normative JSON schema shown below.

```
{
    "$schema":"http://json–schema.org/draft–04/schema#",
    "title": "End of Stream",
    "description": "Normative schema for end_of_stream message .",
    "type": "object",
    "properties": {
      "headers": {
        "type": "string"
      }
    },
    "required": []
}
```

8.4.6 Segment cancel (client → server)

This message represents a client request for the server to cancel the outstanding push transaction over a given WebSocket stream. If no outstanding push transaction is in effect, this message will have no effect. In the case where the cancel is to take effect immediately (signalled by the "immediate" parameter in the description of this message), the server should cancel the on-going pushed segment and all pushed segments that have been scheduled by the server. In the case where the cancel is not immediate, the server should continue to send the next pushed segment, and cancel all other scheduled segments.

An optional headers field may be included to provide additional information to the server to aide in processing the message.

— Message name: segment_cancel

— Supplied arguments

Parameter name	Type	Cardinality	Description
`immediate`	Boolean	1	If true, the client indicates that it would like the server to stop transmission immediately. If false, the client indicates it would like the server to complete transmission of the currently pushed segment (if any) before cancelling the transaction.
`headers`	String	0..1	Contains a CRLF separated set of HTTP 1.1 conformant header fields that apply to the request corresponding to this stream.

The supplied arguments shown above shall be JSON encoded conforming to normative JSON schema shown below.

```
{
  "$schema":"http://json–schema.org/draft–04/schema#",
  "title": "Segment Cancel",
  "description": "Normative schema for segment_cancel message .",
  "type": "object",
  "properties": {
    "immediate": {
        "type": "boolean"
    },
    "headers": {
        "type": "string"
    }
  },
  "required": ["immediate"]
}
```

8.5 MPEG-DASH sub-protocol registration

RFC 6455 requires that sub-protocols be registered with the IANA. The registry requires the following information:

Subprotocol-Identifier: “2016.serverpush.dash.mpeg.org”

Subprotocol Common Name: “MPEG-DASH-ServerPush-23009-6-2017”

Subprotocol Definition: Refers to this document.

Annex A
(informative)

Considered use cases

A.1 Use case 1: Low latency live streaming

In this use case, a viewer begins playback of a live DASH presentation, with the intention of viewing the presentation as close to live as possible.

To fulfil this usage, the DASH media has been prepared to make it possible to achieve low live latency by reducing the segment size. While reducing segment size makes it possible to access content closer to the live point, it is important to ensure the number of server transactions does not increase at the same time.

In addition, small segments are generally not efficient over TCP/IP because of slow start behaviour and it is important to provide methods to efficiently use the network link while achieving low latency live performance.

A.2 Use case 2: Fast start time

In this use case, a viewer begins playback of a VOD DASH presentation and would like the presentation to start as quickly as possible.

A.3 Use case 3: Web browser playback

A viewer begins playback as described in use case 1 or 2, using a standards-based web browser. This web browser supports WebSocket and may also support HTTP/2, although there is no way for the media application to know whether HTTP/2 is supported by the browser directly.

A.4 Use case 4: HTTP-compatible full duplex protocol not supported by client

A viewer begins playback of a DASH presentation. The DASH client does not support a push-based protocol, although in this case, the server does. The playback session is initiated and operates smoothly using HTTP/1.1 as a transport.

A.5 Use case 5: HTTP-compatible full duplex protocol not supported by server

A viewer begins playback of a DASH presentation. The server does not support a push-based protocol, although in this case, the DASH client does. The playback session is initiated and operates smoothly using HTTP/1.1 as a transport.

Annex B
(informative)

System architecture for HTTP/2

The architecture of an end-to-end video streaming system over HTTP/2 is shown in Figure B.1. There are three major system components: 1) the origin server to host the video assets for streaming, which is an HTTP/2 enabled web server deployed with one or more video push strategies; 2) the DASH client to receive and play back the video stream, which consists of a HTTP/2 enabled web browser and a video player; and (3) a content distribution network (CDN) in between the client and origin, which consists of HTTP/2 enabled web cache servers, deployed with one or more push strategies.

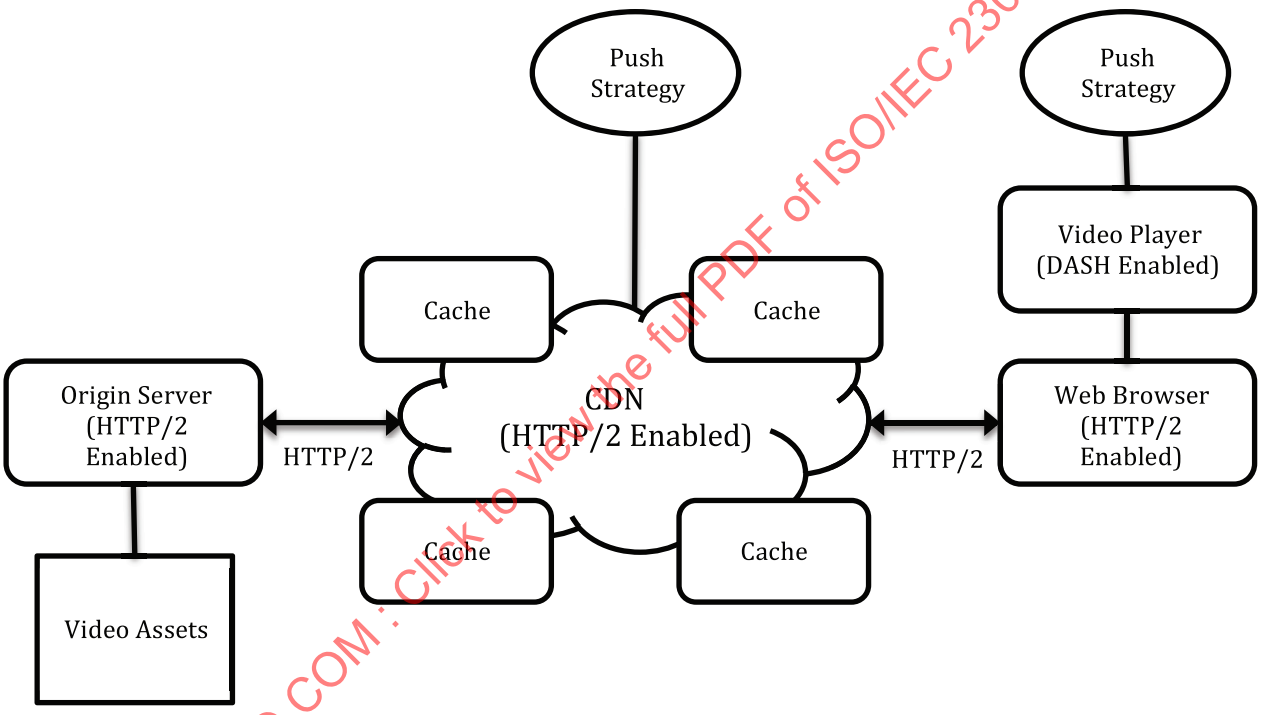
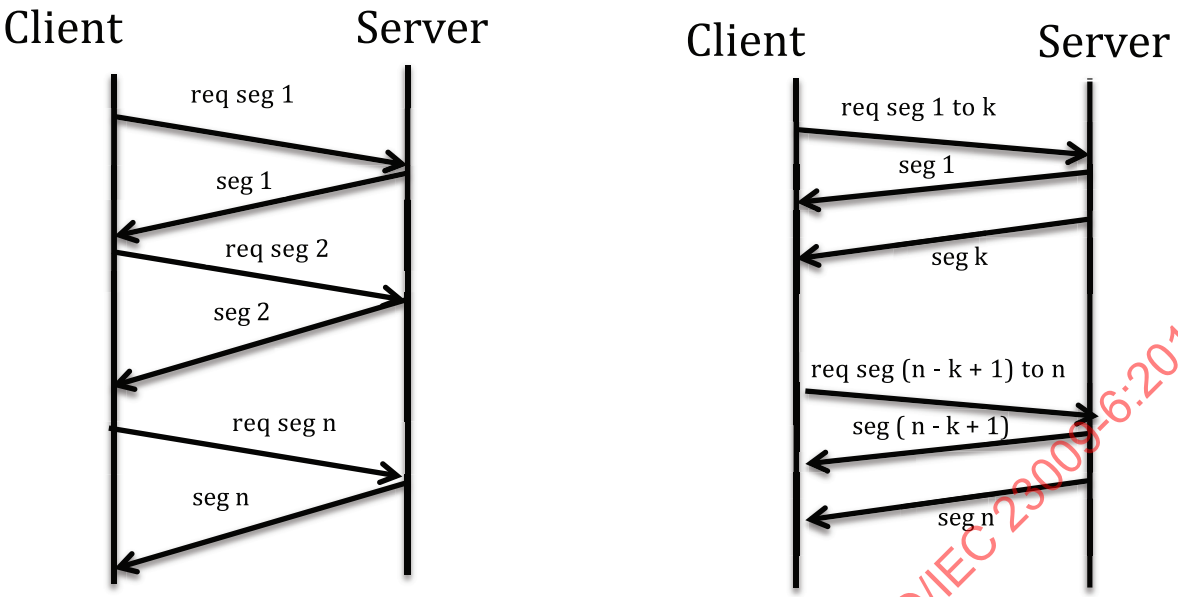


Figure B.1 — System architecture of HTTP/2 DASH streaming

In this system, there are two HTTP/2 persistent connections, one between the client and the CDN and one between the CDN and the origin server. In addition, a tunnelled HTTP/2 connection may also be established between the client and origin, for live streaming that requires low latencies. Unlike HTTP 1.0/1.1 streaming, in HTTP/2, the server (origin or cache) can actively push segments to the client (or the CDN) as soon as they are generated, in addition to the resources that have been explicitly requested by the client (or the CDN) (see Figure B.2).



a) Regular HTTP **b) HTTP/2 server push**

Figure B.2 — Pushing segments using HTTP/2 server push

Annex C
(informative)

Examples of HTTP/2 client/server behaviour

C.1 Example of segment push using “push-next”

In this example, a client requests that the server pushes the next two segments after the one initially requested.

Request [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment1
       accept-push-policy = "urn:mpeg:dash:serverpush:2017:push-next";2;q=1.0
```

Response [Stream ID = 1]:

```
PUSH_PROMISE
       Stream ID = 2
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment2
PUSH_PROMISE
       Stream ID = 4
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment3
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
       push-policy = "urn:mpeg:dash:serverpush:2017:push-next";2
DATA
        + END_STREAM
{binary data for segment 1}
```

Response [Stream ID = 2]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
DATA
        + END_STREAM
{binary data for segment 2}
```

Response [Stream ID = 4]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
DATA
        + END_STREAM
{binary data for segment 3}
```

C.2 Example of segment push using “push-template”

In this example, a client requests that the server pushes a set of segments based on a provided push template.

Request [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment1
       accept-push-policy = "urn:mpeg:dash:serverpush:2017:push-template";'../
rendition1/segment{}':{2,3};q=1.0
```

Response [Stream ID = 1]:

```
PUSH_PROMISE
       Stream ID = 2
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment2
PUSH_PROMISE
       Stream ID = 4
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment3
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
       push-policy = "urn:mpeg:dash:serverpush:2017:push-template";'../
rendition1/segment{}':{2,3}
DATA
        + END_STREAM
{binary data for segment 1}
```

Response [Stream ID = 2]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
DATA
        + END_STREAM
{binary data for segment 2}
```

Response [Stream ID = 4]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
DATA
        + END_STREAM
{binary data for segment 3}
```

C.3 Example of initiating a push request with a server that does not support push

In this example, a client requests that the server pushes the next two segments after the one initially requested. The server is an older server that does not understand push directives. The server does not return a push acknowledgement or promise any additional segments.

Request [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment1
       accept-push-policy = "urn:mpeg:dash:serverpush:2017:push-next";2;q=1.0
```

Response [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
DATA
        + END_STREAM
{binary data for segment 1}
```

In this alternative example, the server *does* understand the push directive, but is not configured to deliver pushed segments or has otherwise elected not to honour the push request. The server explicitly signals this with a push acknowledgment of "urn:mpeg:dash:serverpush:2017:push-none".

Request [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment1
       accept-push-policy = "urn:mpeg:dash:serverpush:2017:push-next";2;q=1.0
```

Response [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
       push-policy = "urn:mpeg:dash:serverpush:2017:push-none"
DATA
        + END_STREAM
{binary data for segment 1}
```

C.4 Example of cancelling a push request

In this example, a client requests that the server pushes the next two segments after the one initially requested. The client receives the initial segment, as well as the next one. The client cancels the stream associated with the third segment, ending the push transaction. This example is representative of what may occur if the client decides to switch representations, i.e. an adaptive bitrate switch, after issuing a push request or if an MPD update makes the previously requested segments unnecessary.

Request [Stream ID = 1]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment1
```

```
       accept-push-policy = "urn:mpeg:dash:serverpush:2017:push-next";2;q=1.0
```

Response [Stream ID = 1]:

```
PUSH_PROMISE
       Stream ID = 2
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment2
PUSH_PROMISE
       Stream ID = 4
       + END_HEADERS
       :method = GET
       :scheme = http
       :path = /example/rendition1/segment3
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
       push-policy = "urn:mpeg:dash:serverpush:2017:push-next";2
DATA
        + END_STREAM
{binary data for segment 1}
```

Response [Stream ID = 2]:

```
HEADERS
       + END_STREAM
       + END_HEADERS
       :status = 200
DATA
        + END_STREAM
{binary data for segment 2}
```

Request [Stream ID = 4]:

```
RST_STREAM
       Error Code = CANCEL
```

Annex D
(informative)

Examples of WebSocket client/server behaviour

D.1 Example of a client requesting an MPD

In this example, a client requests that the server sends the specified MPD.

Client Request:

```
STREAM_ID : 1
MSG_CODE: 1
EXT_LENGTH: 27
EXT: {"mpd_uri":"./example.mpd"}
```

Server Response:

```
STREAM_ID : 1
MSG_CODE: 3
EXT_LENGTH: 0
{binary data with example.mpd}
```

D.2 Example of a client requesting a segment, using a push directive

In this example, the client requests a segment, indicating that the server should push the next two segments after the one initially requested.

Client Request:

```
MSG_CODE: 2
EXT_LENGTH: 104
EXT: {"segment_uri":"./rep1/segment1.mp4","push_directive":["urn:mpeg:dash:serverpush:2017
:push-next;2;q=1.0"]}
```

Server Response:

```
STREAM_ID : 1
MSG_CODE: 4
EXT_LENGTH: 56
EXT: {"push_ack":["urn:mpeg:dash:serverpush:2017:push-next;2"]}
{binary data with segment1.mp4}

STREAM_ID : 1
MSG_CODE: 4
EXT_LENGTH: 0
{binary data with segment2.mp4}

STREAM_ID : 1
MSG_CODE: 4
EXT_LENGTH: 0
{binary data with segment3.mp4}
```