Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2015

A. Begen
Cisco
K. Streeter
Adobe Systems Incorporated
I. Bouazizi
Samsung Research
F. Denoual
Canon Research Centre France

October 27, 2014

MPEG DASH Requirements for a webpush Protocol
draft-begen-webpush-dash-reqs-00

Abstract

This draft presents two of the ongoing core experiments for the amendment of the MPEG Dynamic Adaptive Streaming over HTTP (DASH) specification and discusses some requirements for a webpush protocol in the context of these core experiments.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 30, 2015.

Copyright Notice

Table of Contents

1. Introduction

HTTP adaptive streaming (HAS) has become the technology of choice for delivering video content over the Internet. HAS will play an increasingly important role in delivering video to the primary and secondary screens.

In HAS, streaming clients are offered multiple representations of the same content. Clients independently choose a segment (i.e., a piece of the content) belonging to a representation to fetch from the content delivery network, a choice that is made at every switching boundary (usually every 2-10 seconds). The choice is based on a number of parameters, including the network throughput observed by the client.

To date they have been several different HAS implementations from different vendors. To achieve interoperability, MPEG has started working on a specification in 2010 and published the first edition of the Dynamic Adaptive Streaming over HTTP specification in 2012. Earlier this year, the second edition has been published [DASH-Part1].

To add new features and capabilities to the DASH applications, the DASH subgroup is currently running a number of core experiments (CE) [DASH-CE]. Two of these CEs are (i) DASH over Full Duplex HTTP-based Protocols (FDH) and (ii) Server and Network Assisted DASH (SAND). In these CEs, there is a need for a webpush protocol. This draft discusses some requirements for a such protocol to be used in DASH applications.

2. DASH Requirements in the Core Experiments

2.1. Requirements from the FDH CE

The FDH CE is investigating the problem of delivering media segments with shorter delay and/or reducing the request processing on the HTTP server. The technologies considered are HTTP/2 and WebSockets.

The main idea is that once a client is interested in streaming a particular content (e.g., a live channel), it may be beneficial not to send an individual GET request for each segment to the network. The network may keep sending segments once they become available to the client for a predetermined amount of time or until the client tells the server to stop.

We realize that a content delivery network or an operator network can consist of both HTTP/2 enabled servers and HTTP 1.1 servers as well as proxy servers, some of which could have been implemented improperly. An important requirement for us is that the push feature should not cause any caching inconsistency or reduce caching efficiency. Note that not all clients streaming the same content will necessarily use the push feature. Some may continue fetching each segment separately (the conventional way), some may ask for the push of the next k segments from the server or some may ask for push till a further command.

An important issue is to decide how the clients will be able to ask for not one but multiple segments in a single GET request, i.e., specifying a list of segments or potentially a pattern that includes wildcards or any other agreeable formats. At least two options are considered: a modified URL scheme and the use of an extension header. In both cases modifications are expected on the origin server and possibly the cache servers (We will likely need a special handler on the servers). We would like to get advice on any of these technologies taking into account among others backward-compatibility, efficiency, caching issues, processing load, extensibility and likelihood of implementation.

The DASH clients may run in browsers, mobile applications, on personal computers or other connected devices. Typically it cannot be assumed that in all applications a DASH client will have access to the HTTP stack in a given platform. Such constraints should be taken into account when looking for a broadly acceptable and highly-scalable solution.

2.2. Requirements from the SAND CE

The SAND CE has the goal of establishing a bi-directional messaging plane between the clients and other so-called Dash-aware Network Elements (DANE). The DANEs may also communicate with each other. Common DANEs include proxies and caches as well as analytics servers and other network devices. On the direction from a DANE to a DASH client (or another DANE), the messages can carry any kind of operational information and/or assistance information so that the DASH client (or the DANE) can take a proper action. On the direction from a DASH client to a DANE, the messages are expected to carry metrics and status information.

A simple use case for sending an operational information to a DASH client is to ask the client to switch to a particular CDN provider. Another use case is to ask the DASH client to fetch a particular representation, simply because that representation's segments are already cached at the edge of the network. Yet other use cases may include network status information such as bitrate guarantees, delays, etc.

An obvious choice for the protocol to carry these SAND messages back and forth is HTTP. The DASH clients can send their metrics and status messages upon demand or periodically to a pre-designated server. However, sending messages in the other direction (from a DANE to a DASH client) may be more challenging since DASH clients may or may not be expecting such a message. It is important to note that we do not want to put time-sensitive client-specific feedback messages on top of the HTTP responses containing non-time-critical and non-client-specific media segments.

To enable bi-directional communication between a DASH client and a DANE, we can also use WebSockets or the HTTP/2 PUSH technologies. However, we consider a more lightweight protocol that will scale to large populations when sending a common message or individual messages.

3. Guidance from the IETF

We would like to ask the webpush WG to consider media delivery use cases as part of the webpush activity, including the delivery of MPEG DASH content. We also would like to ask the webpush WG to advise us on best practices on using header and URL based signaling for push behavior, and get feedback on proper message channels for SAND.

4. Security Considerations

There are no security considerations.

5. IANA Considerations

There are no IANA considerations.

6. Informative References

[DASH-Part1] Available at: http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html, , "ISO/IEC 23009-1:2014: Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats (2nd edition)", May 2014.

[DASH-CE] Available at: http://wg11.sc29.org/doc_end_user/current_meeting_intermediate.php?id_meeting=162, , "Descriptions of core experiments on DASH amendment (w14858)", Oct. 2014.

Authors' Addresses

Ali Begen
Cisco
181 Bay Street
Toronto, ON M5J 2T3
Canada

EMail: abegen@cisco.com

Kevin Streeter
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110
USA

EMail: kstreete@adobe.com

Imed Bouazizi
Samsung Research
1301 E. Lookout Dr.
Richardson, TX 75082
USA

EMail: i.bouazizi@sta.samsung.com

Franck Denoual
Canon Research Centre France
Rue de la Touche Lambert
Cesson-Sevigne 35517
France

EMail: franck.denoual@crf.canon.fr

WEBPUSH
Internet-Draft
Intended status: Standards Track
Expires: April 11, 2015

M. Thomson
Mozilla

October 8, 2014

Generic Event Delivery Using HTTP Push
draft-thomson-webpush-http2-01

Abstract

A simple protocol for the delivery of realtime events to clients is described. This scheme uses HTTP/2 push.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 11, 2015.

Table of Contents

1. Introduction

Mobile computing devices are increasingly relied upon for a great many applications. Mobile devices typically have limited power reserves, so finding more efficient ways to serve application requirements is an important part of any mobile platform.

One significant contributor to power usage mobile devices is the radio. Radio communications consumes a significant portion of the energy budget on a wirelessly connected mobile device.

Many applications require continuous access to network communications so that real-time events - such as incoming calls or messages - can be conveyed (or "pushed") to the user in a timely fashion. Uncoordinated use of persistent connections or sessions from multiple applications can contribute to unnecessary use of the device radio, since each independent session independently incurs overheads. In particular, keep alive traffic used to ensure that middleboxes do not prematurely time out sessions, can result in significant waste. Maintenance traffic tends to dominate over the long term, since events are relatively rare.

Consolidating all real-time events into a single session ensures more efficient use of network and radio resources. A single service consolidates all events, distributing those events to applications as they arrive. This requires just one session, avoiding duplicated overhead costs.

The Web Push API [API] describes an API that enables the use of a consolidated push service from web applications. This expands on that work by describing a protocol that can be used to:

- request the delivery of an event to a device,
- register a new device,
- create new event delivery channels, and
- monitor for new events.

This is intentionally split into these two categories because requesting the delivery of events is required for immediate use by the Web Push API. The registration, management and monitoring functions are currently fulfilled by proprietary protocols; these are adequate, but do not offer any of the advantages that standardization affords.

The monitoring function described in this document is intended to be replaceable, enabling the use of monitoring schemes that are better optimized for the network environment and the device. For instance, using notification systems like the GSM Short Message Service (SMS) can take advantage of the native paging capabilities of a cellular network, avoiding the ongoing maintainence cost of a persistent TCP connection.

This document intentionally does not describe how a push server is discovered. Discovery of push servers is left for future efforts, if it turns out to be necessary at all. Devices are expected to be configured with a push server URL.

Similarly, discovery of support for and negotiation of use of alternative monitoring schemes is left to documents that extend this basic protocol.

1.1. Conventions and Terminology

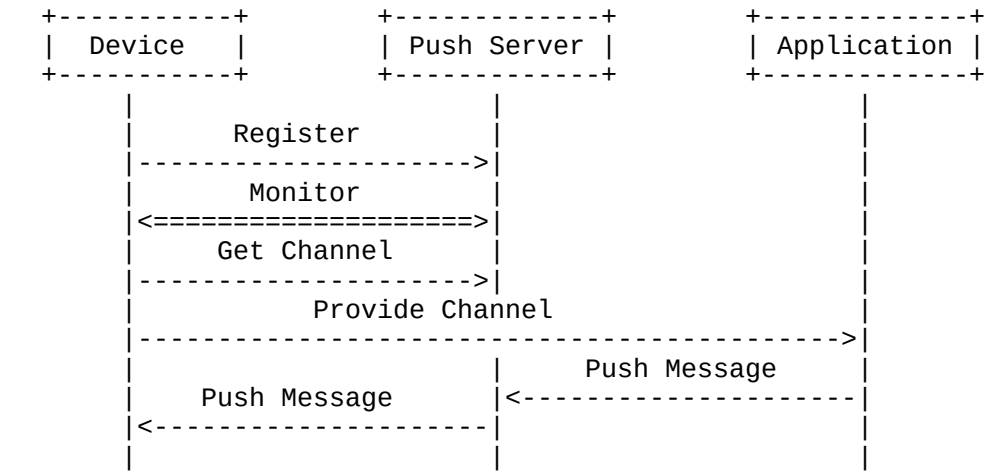
In cases where normative language needs to be emphasized, this document falls back on established shorthands for expressing interoperability requirements on implementations: the capitalized words "MUST", "MUST NOT", "SHOULD" and "MAY". The meaning of these is described in [RFC2119].

This document will use the terminology from [API], though "application" will be used in preference to "webapp", since the described protocols are not restricted to web use. This document

introduces the term "device", which refers to the consumer of push messages.

2. Overview

A general model for push services includes three basic actors: a device, a push server, and an application.



At the very beginning of the process, the device registers with the push server. This establishes a shared session between the device and push server that will be used to aggregate push messages from all applications that the device interacts with.

The registration response includes details on how the device is expected to monitor for incoming push messages. This document describes one such mechanism, though more efficient means of monitoring could be optionally defined (and this is expressly permitted).

A registration after creation has no channels associated with it. New channels can be requested by the device and then distributed to applications. It is expected that devices will distribute a different channel to each application, with the potential for multiple channels being provided to the same application.

Applications use channels to deliver push messages to devices, via the push server.

Both registrations and channels have a limited lifetime. These will need to be refreshed or replaced over time.

3. Delivering Push Messages

A push channel is identified with an HTTP URI [RFC7230]. An application can request the delivery of a push message by sending an HTTP PUT request to this URI, including the push message in the body of the request.

A push server can acknowledge the end-to-end delivery of a push message by responding with a 200 (OK) status code. A push server that stores the message for later delivery (see Section 7) could respond with a 202 (Accepted) status code to indicate that the message was stored, but not delivered.

4. Registering

A device that wishes to establish a new or replacement registration sends an HTTP POST request to its configured push server URL. The request contains no entity body.

The push server creates a new registration in response to this request, creating two new resources and allocating an HTTP URI for each. These URIs are included in link relations [RFC5988] that are included in Link header fields in the response.

monitor: A link relation of type "...:push:monitor" includes the URL of a resource that the device can monitor for events. Monitoring is described in Section 6.

channel: A link relation of type "...:push:channel" includes a URL of a resource where the device can create new channels. Creating channels is described in Section 5.

The push server includes the "monitor" link relation in a Location header field.

The push server MUST include expiration information in the response to this request in either the Expires header field, or by setting a "max-age" parameter on a Cache-Control header field. The Cache-Control header field MUST include the "private" directive [RFC7235].

The push server SHOULD also provide the "channel" link and expiration information in response to requests to the "monitor" resource.

A device MUST support the 307 (Temporary Redirect) status code [RFC7231], which can be used by a push server to redistribute load at the time a registration is created.

5. Channels

A client sends a POST request to the "channel" resource to create a new channel.

A response with a 201 status code includes the channel URI in the Location header field.

A channel can expire. Servers indicate this using the Expires header field, or by setting a "max-age" parameter on a Cache-Control header field.

A client can explicitly delete a channel by sending a DELETE request to channel URI.

6. Monitoring and Receiving Push Messages

A device monitors for new events by making a GET request to the monitor resource. The server does not respond to these request, it instead uses server push [I-D.ietf-httpbis-http2] to send the contents of push messages as applications send them.

Each push message consists of a synthesized GET request to the channel URI that was the target of the push. The response body is the entity body from the PUT request.

A device can request the monitor resource immediately by including a Prefer header field [RFC7240] with a "wait" parameter set to "0". This allows clients to rapidly check for any missed messages. Clients can check the status of individual channels by sending GET requests to the channel URI.

A server that wishes to redistribute load can do so using the alternative services mechanisms that are part of HTTP/2 [I-D.ietf-httpbis-alt-svc]. The ALTSVC frame type allows for redistribution of load whilst retaining the same monitor resource. Once a device has established a replacement connection, it can notify the server of imminent shutdown using a GOAWAY frame, which allows the server to respond to the long-standing GET request and gracefully shut down the connection. This allows for seamless migration between servers.

7. Store and Forward Operation

Push servers are not obligated to store messages for any time. If a client is not actively monitoring for push messages, messages can be lost.

Push servers can store messages for some time to allow for limited recovery from transient faults. If a message is stored, but not delivered, the push server can indicate the probable duration of storage by including expiration information in the response to the push request.

Messages that were stored and not delivered to a client MAY be delivered when a client commences monitoring. These messages should include a Last-Modified header field. If a server stores push messages, a GET request to a channel URI returns the last message sent by an application to that channel.

Push servers that store push messages might need to limit the size of push messages to avoid being subject to overloading. Push servers that don't store can stream the payload of push messages to devices. This can use HTTP/2 flow control to limit the state commitment this requires. However, push servers MAY place an upper limit on the size of push messages that they permit.

8. IANA Considerations

TODO: register link relation types, as necessary.

9. Security Considerations

This protocol MUST use HTTP over TLS [RFC2818]; this includes any communications between device and push server, plus communications between the application and the push server. This provides confidentiality and integrity protection for registrations and push message.

9.1. Confidentiality from Push Server Access

The protection afforded by TLS does not protect content from the push server. A push server is able to see and modify the content of the messages.

Applications are able to provide additional confidentiality, integrity or authentication mechanisms within the push message itself. The originating application server and the device are frequently just different instances of the same application, this does not require standardization. The process of registering a channel endpoints provides a convenient medium for key agreement.

In particular, the W3C Web Push API requires that each push channel created by the browser be bound to a browser generated encryption key. Pushed messages are authenticated and decrypted by the browser

before delivery. This ensures that the push server is unable to examine the contents of push messages.

The public key for a channel ensures that applications using a channel can identify messages from unknown sources and discard them. This depends on the public key only being disclosed to entities that are authorized to send messages on the channel. The push server does not require access to this public key.

9.2. Privacy Considerations

Push message confidentiality does not ensure that the identity of who is communicating and when they are communicating is protected. However, the amount of information that is exposed can be limited.

The identifiers used by this protocol provide some ability to correlate communications for a given device, either across applications or over time. Most important is that communications for a given device not be able to be correlated between different application usages, or between different times.

Channel URIs established by the same device MUST NOT include any information that allows them to be correlated with other channels or the device registration. The push server is the only entity that needs to be able to correlate channel URIs with device registrations. Note that this can't prevent the use of traffic analysis in performing correlation.

A device MUST be able to create new registrations at any time. Identifiers for new registrations MUST NOT include any information that allows them to be correlated with other registrations from the same device or user.

9.3. Denial of Service Vectors

This protocol does not specify a single authorization framework for managing access to push servers, either by devices or applications. Thus, there is a very real possibility that this could be exploited to mount denial of service attacks on the push server. Push servers MAY choose to authorize requests based on any HTTP-compatible means available, of which there are numerous options.

Discarding unwanted messages at the device based on message authentication doesn't protect against a denial of service attack on the device. Even a relatively small number of message can cause devices to exhaust batteries. Limiting the number of entities with access to push channels limits the number of entities that can generate value push requests of the push server. An application can

do this by controlling the distribution of channel URIs to authorized entities.

Only the push server can make this denial of service protection possible. A push server MUST generate channel URI that are extremely difficult to guess. Encoding a large amount of random entropy (at least 128 bits) in the URI path is one technique for ensuring that channel URIs are able to act as bearer tokens.

A malicious application can use the greater resources of a push server to mount a denial of service attack on devices. Push servers SHOULD limit the rate at which push messages are sent to devices.

Conversely, a push server is also able to deny service to devices. Intentional failure to deliver messages is difficult to distinguish from faults, which might occur due to transient network errors, interruptions in device availability, or genuine service outages. Applications that rely on reliable message delivery need to provide means of recovering from occasional failures that do not rely on push notifications.

9.4. Logging Exposure

Server request logs can reveal registration and channel URIs. Acquiring a registration URI permits the creation of new channels and the receipt of messages. Acquiring either URI permits the generation of push messages. Logging could also reveal relationships between different channel URIs for the same registration, or between different registrations for the same device.

End-to-end confidentiality mechanisms, such as those in [API], prevent an entity with a registration URI from learning the contents of push messages. In both cases, push messages that are not successfully authenticated will not be delivered by the API, but this can present a denial of service risk. Limitations on log retention and strong access control mechanisms can ensure that these URIs are not learned.

10. References

10.1. Normative References

[I-D.ietf-httpbis-alt-svc]
Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-01 (work in progress), April 2014.

[I-D.ietf-httpbis-http2]
Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-12 (work in progress), April 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7235] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

[RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240, June 2014.

10.2. Informative References

[API] Sullivan, B. and E. Fullea, "Web Push API", Editor's Draft push-api, May 2014, <https://w3c.github.io/push-api/index.html>.

Author's Address

Martin Thomson
Mozilla
331 E Evelyn Street
Mountain View, CA 94041
US

Email: martin.thomson@gmail.com