WEBPUSH
Internet-Draft
Intended status: Standards Track
Expires: May 26, 2016

M. Thomson
Mozilla
E. Damaggio
B. Raymor, Ed.
Microsoft

November 23, 2015

Generic Event Delivery Using HTTP Push
draft-ietf-webpush-protocol-02

Abstract

A simple protocol for the delivery of realtime events to user agents is described. This scheme uses HTTP/2 server 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 May 26, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

Many applications on mobile and embedded devices require continuous access to network communications so that real-time events - such as incoming calls or messages - can be delivered (or "pushed") in a timely fashion. These devices typically have limited power reserves, so finding more efficient ways to serve application requirements greatly benefits the application ecosystem.

One significant contributor to power usage is the radio. Radio communications consume a significant portion of the energy budget on a wireless device.

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 W3C 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 a push message to a user agent,
- create new push message delivery subscriptions, and
- monitor for new push messages.

Requesting the delivery of events is particularly important for the Web Push API. The subscription, management and monitoring functions are currently fulfilled by proprietary protocols; these are adequate, but do not offer any of the advantages that standardization affords.

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

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 defines the following terms:

application: Both the sender and ultimate consumer of push messages. Many applications have components that are run on a user agent and other components that run on servers.

application server: The component of an application that runs on a server and requests the delivery of a push message.

push message subscription: A message delivery context that is established between the user agent and the push service and shared with the application server. All push messages are associated with a push message subscription.

push message subscription set: A message delivery context that is established between the user agent and the push service that collects multiple push message subscriptions into a set.

push message: A message sent from an application server to a user agent via a push service.

push message receipt: A message delivery confirmation sent from the push service to the application server.

push service: A service that delivers push messages to user agents.

user agent: A device and software that is the recipient of push messages.

Examples in this document use the HTTP/1.1 message format [RFC7230]. Many of the exchanges can be completed using HTTP/1.1, where HTTP/2 is necessary, the more verbose frame format from [RFC7540] is used.

2. Overview

A general model for push services includes three basic actors: a user agent, a push service, and an application (server).

```
    +-------+           +--------------+       +-------------+
    |  UA   |           | Push Service |       | Application |
    +-------+           +--------------+       +-------------+
        |                      |                      |
        |      Subscribe       |                      |
        |--------------------->|                      |
        |       Monitor        |                      |
        |<====================>|                      |
        |                      |                      |
        |          Distribute Push Resource           |
        |-------------------------------------------->|
        |                      |                      |
        :                      :                      :
        |                      |     Push Message     |
        |    Push Message      |<---------------------|
        |<---------------------|                      |
        |                      |                      |
```

At the very beginning of the process, a new message subscription is created by the user agent and then distributed to its application server. This subscription is the basis of all future interactions between the actors.

To offer more control for authorization, a message subscription is modeled as two resources with different capabilities:

- A subscription resource is used to receive messages from a subscription and to delete a subscription. It is private to the user agent.

- A push resource is used to send messages to a subscription. It is public and shared by the user agent with its application server.

It is expected that a unique subscription will be distributed to each application; however, there are no inherent cardinality constraints in the protocol. Multiple subscriptions might be created for the same application, or multiple applications could use the same subscription. Note however that sharing subscriptions has security and privacy implications.

Subscriptions have a limited lifetime. They can also be terminated by either the push service or user agent at any time. User agents and application servers must be prepared to manage changes in subscription state.

2.1. HTTP Resources

This protocol uses HTTP resources [RFC7230] and link relations [RFC5988]. The following resources are defined:

push service: This resource is used to create push message subscriptions (see Section 4). A URL for the push service is configured into user agents.

push message subscription: This resource provides read and delete access for a message subscription. A user agent receives push messages (Section 7) using a push message subscription. Every push message subscription has exactly one push resource associated with it.

push message subscription set: This resource provides read and delete access for a collection of push message subscriptions. A user agent receives push messages (Section 7.1) for all the push message subscriptions in the set. A link relation of type "urn:ietf:params:push:set" identifies a push message subscription set.

push: A push resource is used by the application server to request the delivery of a push message (see Section 6). A link relation of type "urn:ietf:params:push" is used to identify a push resource.

push message: A push message resource is created to identify push messages that have been accepted by the push service. The push message resource is also used to acknowledge receipt of a push message.

receipt subscribe: A receipt subscribe resource is used by an application server to create a receipt subscription (see Section 5). A link relation of type "urn:ietf:params:push:receipt" is used to identity a receipt subscribe resource.

receipt subscription: An application server receives delivery confirmations (Section 6.1) for push messages using a receipt subscription.

3. Connecting to the Push Service

The push service shares the same default port number (443/TCP) with HTTPS, but MAY also advertise the IANA allocated TCP System Port 1001 using HTTP alternative services [I-D.ietf-httpbis-alt-svc]:

While the default port (443) offers broad reachability characteristics, it is most often used for web browsing scenarios with a lower idle timeout than other ports configured in middleboxes. For webpush scenarios, this would contribute to unnecessary radio communications to maintain the connection on battery-powered devices.

Advertising the alternate port (1001) allows middleboxes to optimize idle timeouts for connections specific to push scenarios with the expectation that data exchange will be infrequent.

Middleboxes SHOULD comply with REQ-5 in [RFC5382] which requires that "the value of the ’established connection idle-timeout’ MUST NOT be less than 2 hours 4 minutes".

4. Subscribing for Push Messages

A user agent sends a POST request to its configured push service resource to create a new subscription.

```
POST /subscribe/ HTTP/1.1
Host: push.example.net
```

A response with a 201 (Created) status code includes a URI for a new push message subscription resource in the Location header field.

The push service MUST provide a URI for the push resource corresponding to the push message subscription using a link relation of type "urn:ietf:params:push".

The push service MUST provide a URI for a receipt subscribe resource in a link relation of type "urn:ietf:params:push:receipt".

An application-specific method is used to distribute the push and receipt subscribe URIs to the application server. Confidentiality protection and application server authentication MUST be used to ensure that these URIs are not disclosed to unauthorized recipients (see Section 9.3).

The push service MAY provide a URI for a subscription set resource in a link relation of type "urn:ietf:params:push:set". If provided, the push service supports subscription sets. If available, the user agent SHOULD use the subscription set to receive push messages rather than individual push message subscriptions.

The push service MAY return new subscription sets in response to different subscription requests from the same user agent. This allows the push service to control the grouping of the push message subscriptions.

```
   HTTP/1.1 201 Created
   Date: Thu, 11 Dec 2014 23:56:52 GMT
   Link: </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
           rel="urn:ietf:params:push"
   Link: </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
           rel="urn:ietf:params:push:receipt"
   Link: </set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
           rel="urn:ietf:params:push:set"
   Location: https://push.example.net/s/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
```

4.1. Correlating Subscriptions

PENDING: Discuss how two different subscriptions are correlated based on either authentication or connection

5. Subscribing for Push Message Receipts

An application server requests the creation of a receipt subscription by sending a HTTP POST request to the receipt subscribe resource distributed to the application server by a user agent.

```
   POST /receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ HTTP/1.1
   Host: push.example.net
```

A successful response with a 201 (Created) status code includes a URI for the receipt subscription resource in the Location header field.

```
   HTTP/1.1 201 Created
   Date: Thu, 11 Dec 2014 23:56:52 GMT
   Location: https://push.example.net/r/3ZtI4YVNBnUUZhuoChl6omUvG4ZM9mpN
```

An application server that sends push messages to a large population of user agents incurs a significant load if it has to monitor a receipt subscription for each user agent. Reuse of receipt subscriptions is critical in reducing load on application servers. A receipt subscription can be used for all resources that have the same receipt subscribe URI.

A push service SHOULD provide the same receipt subscribe URI to all user agents. Application servers SHOULD reuse receipt subscription URIs if the receipt subscribe URI provided with the push resource is identical to the one used to create the receipt subscription. Checking that the receipt subscribe URI is identical allows the application server to avoid creating unnecessary receipt subscriptions.

6. Requesting Push Message Delivery

An application server requests the delivery of a push message by sending a HTTP request to a push resource distributed to the application server by a user agent. The push message is included in the body of the request.

```
POST /p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
Push-Receipt: https://push.example.net/r/3ZtI4YVNBnUUZhuoChl6omUvG4ZM9mpN
Content-Type: text/plain;charset=utf8
Content-Length: 36

iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

A 201 (Created) response indicates that the push message was accepted. A URI for the push message resource that was created in response to the request is included in the Location header field. This does not indicate that the message was delivered to the user agent.

```
   HTTP/1.1 201 Created
   Date: Thu, 11 Dec 2014 23:56:55 GMT
   Location: https://push.example.net/d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
```

A push service MAY return a 429 (Too Many Requests) status code [RFC6585] when an application server has exceeded its rate limit for push message delivery to a push resource. The push service SHOULD also include a Retry-After header [RFC7231] to indicate how long the application server is requested to wait before it makes another request to the push resource.

A push service MAY return a 413 (Payload Too Large) status code [RFC7231] in response to requests that include an entity body that is too large. Push services MUST NOT return a 413 status code in responses to an entity body that is 4k (4096 bytes) or less in size.

6.1. Requesting Push Message Receipts

An application server can use the Push-Receipt header field to request a confirmation from the push service when a push message is delivered and acknowledged by the user agent. The Push-Receipt header field is a URI-Reference as defined in Section 2.7 of [RFC7230].

```
Push-Receipt = URI-reference
```

The application sets the Push-Receipt header field value to a receipt subscription URI. This receipt subscription resource MUST be created from the same receipt subscribe resource which was returned with the push message subscription response (see Section 4).

6.2. Push Message Time-To-Live

A push service can improve the reliability of push message delivery considerably by storing push messages for a period. User agents are often only intermittently connected, and so benefit from having short term message storage at the push service.

Delaying delivery might also be used to batch communication with the user agent, thereby conserving radio resources.

Some push messages are not useful once a certain period of time elapses. Delivery of messages after they have ceased to be relevant is wasteful. For example, if the push message contains a call notification, receiving a message after the caller has abandoned the call is of no value; the application at the user agent is forced to suppress the message so that it does not generate a useless alert.

An application server can use the TTL header field to limit the time that a push message is retained by a push service. The TTL header field contains a value in seconds that describes how long a push message is retained by the push service.

```
TTL = 1*DIGIT
```

Once the Time-To-Live (TTL) period elapses, the push service MUST NOT attempt to deliver the push message to the user agent. A push service might adjust the TTL value to account for time accounting errors in processing. For instance, distributing a push message within a server cluster might accrue errors due to clock skew or propagation delays.

A push service is not obligated to account for time spent by the application server in sending a push message to the push service, or delays incurred while sending a push message to the user agent. An application server needs to account for transit delays in selecting a TTL header field value.

Absence of the TTL header field is interpreted as equivalent to a zero value. A Push message with a zero TTL is immediately delivered if the user agent is available to receive the message. After delivery, the push service is permitted to immediately remove a push message with a zero TTL. This might occur before the user agent acknowledges receipt of the message by performing a HTTP DELETE on the push message resource. Consequently, an application server cannot rely on receiving acknowledgement receipts for zero TTL push messages.

If the user agent is unavailable, a push message with a zero TTL expires and is never delivered.

A push service MAY choose to retain a push message for a shorter duration than that requested. It indicates this by including a TTL header field in the response that includes the actual TTL. This TTL value MUST be less than or equal to the value provided by the application server.

7. Receiving Push Messages for a Subscription

A user agent requests the delivery of new push messages by making a GET request to a push message subscription resource. The push service does not respond to this request, it instead uses HTTP/2 server push [RFC7540] to send the contents of push messages as they are sent by application servers.

Each push message is pushed as the response to a synthesized GET request in a PUSH_PROMISE. This GET request is made to the push message resource that was created by the push service when the application server requested message delivery. The response headers SHOULD provide a URI for the push resource corresponding to the push message subscription in a link relation of type "urn:ietf:params:push". The response body is the entity body from the most recent request sent to the push resource by the application server.

The following example request is made over HTTP/2.

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /s/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When a push message is sent by an application server, a server push is associated with the initial request. The response includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method        = GET
  :path          = /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
  :authority     = push.example.net

HEADERS      [stream 4] +END_HEADERS
  :status        = 200
  date           = Thu, 11 Dec 2014 23:56:56 GMT
  last-modified  = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control  = private
  :link          = </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                    rel="urn:ietf:params:push"
  content-type   = text/plain;charset=utf8
  content-length = 36

DATA         [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

The push service response for the GET request to the push message subscription resource SHOULD provide a URI for the receipt subscribe resource in a link relation of type "urn:ietf:params:push:receipt".

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :status        = 200
  :link          = </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
                    rel="urn:ietf:params:push:receipt"
```

A user agent can also request the contents of the push message subscription resource immediately by including a Prefer header field [RFC7240] with a "wait" parameter set to "0". In response to this request, the push service MUST generate a server push for all push messages that have not yet been delivered.

A 204 (No Content) status code with no associated server pushes indicates that no messages are presently available. This could be because push messages have expired.

7.1. Receiving Push Messages for a Subscription Set

There are minor differences between receiving push messages for a subscription and a subscripion set.

A user agent requests the delivery of new push messages for a collection of push message subscriptions by making a GET request to a push message subscription set resource. The push service does not respond to this request, it instead uses HTTP/2 server push [RFC7540] to send the contents of push messages as they are sent by application servers.

Each push message is pushed as the response to a synthesized GET request sent in a PUSH_PROMISE. This GET request is made to the push message resource that was created by the push service when the application server requested message delivery. The synthetic request MUST provide a URI for the push resource corresponding to the push message subscription in a link relation of type "urn:ietf:params:push". This enables the user agent to differentiate the source of the message. The response body is the entity body from the most recent request sent to the push resource by an application server.

The following example request is made over HTTP/2.

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When a push message is sent by an application server, a server push is associated with the initial request. The response includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method        = GET
  :path          = /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
  :authority     = push.example.net
  :link          = </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                      rel="urn:ietf:params:push"

HEADERS      [stream 4] +END_HEADERS
  :status        = 200
  date           = Thu, 11 Dec 2014 23:56:56 GMT
  last-modified  = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control  = private
  content-type   = text/plain;charset=utf8
  content-length = 36

DATA         [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

The push service response for the GET request to the push message subscription set resource SHOULD provide a URI for the receipt subscribe resource in a link relation of type "urn:ietf:params:push:receipt".

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :status        = 200
  :link          = </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
                      rel="urn:ietf:params:push:receipt"
```

A user agent can request the contents of the push message subscription set resource immediately by including a Prefer header field [RFC7240] with a "wait" parameter set to "0". In response to this request, the push service MUST generate a server push for all push messages that have not yet been delivered.

A 204 (No Content) status code with no associated server pushes indicates that no messages are presently available. This could be because push messages have expired.

7.2. Acknowledging Push Messages

To ensure that a push message is properly delivered to the user agent at least once, the user agent MUST acknowledge receipt of the message by performing a HTTP DELETE on the push message resource.

```
DELETE /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk HTTP/1.1
Host: push.example.net
```

If the push service receives the acknowledgement and the application has requested a delivery receipt, the push service MUST deliver a success response to the application server monitoring the receipt subscription resource.

If the push service does not receive the acknowledgement within a reasonable amount of time, then the message is considered to be not yet delivered. The push service SHOULD continue to retry delivery of the message until its advertised expiration.

The push service MAY cease to retry delivery of the message prior to its advertised expiration due to scenarios such as an unresponsive user agent or operational constraints. If the application has requested a delivery receipt, then the push service MUST push a failure response with a status code of 410 (Gone) to the application server monitoring the receipt subscription resource.

7.3. Receiving Push Message Receipts

The application server requests the delivery of receipts from the push service by making a HTTP GET request to the receipt subscription resource. The push service does not respond to this request, it instead uses HTTP/2 server push [RFC7540] to send push receipts when messages are acknowledged (Section 7.2) by the user agent.

Each receipt is pushed as the response to a synthesized GET request sent in a PUSH_PROMISE. This GET request is made to the same push message resource that was created by the push service when the application server requested message delivery. A successful response includes a 204 (No Content) status code with no data.

The following example request is made over HTTP/2.

```
HEADERS      [stream 13] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /r/3ZtI4YVNBnUUZhuoChl6omUvG4ZM9mpN
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When the user agent acknowledges the message, the push service pushes a delivery receipt to the application server. A 204 (No Content) status code confirms that the message was delivered and acknowledged.

```
PUSH_PROMISE [stream 13; promised stream 82] +END_HEADERS
  :method        = GET
  :path          = /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
  :authority     = push.example.net

HEADERS      [stream 82] +END_STREAM
                          +END_HEADERS
  :status        = 204
  date           = Thu, 11 Dec 2014 23:56:56 GMT
```

If the user agent fails to acknowledge the receipt of the push message and the push service ceases to retry delivery of the message prior to its advertised expiration, then the push service MUST push a failure response with a status code of 410 (Gone).

8. Operational Considerations

8.1. Load Management

A push service is likely to have to maintain a very large number of open TCP connections. Effective management of those connections can depend on being able to move connections between server instances.

A user agent MUST support the 307 (Temporary Redirect) status code [RFC7231], which can be used by a push service to redistribute load at the time that a new subscription is requested.

A server that wishes to redistribute load can do so using alternative services [I-D.ietf-httpbis-alt-svc]. Alternative services allows for redistribution of load whilst maintaining the same URIs for various resources. User agents can ensure a graceful transition by using the GOAWAY frame once it has established a replacement connection.

8.2. Push Message Expiration

Storage of push messages based on the TTL header field comprises a potentially significant amount of storage for a push service. A push service is not obligated to store messages indefinitely. A push service is able to indicate how long it intends to retain a message to an application server using the TTL header field (see Section 6.2).

A user agent that does not actively monitor for push messages will not receive messages that expire during that interval.

Push messages that are stored and not delivered to a user agent are delivered when the user agent recommences monitoring. Stored push messages SHOULD include a Last-Modified header field (see Section 2.2 of [RFC7232]) indicating when delivery was requested by an application server.

A GET request to a push message subscription resource that has only expired messages results in response as though no push message were ever sent.

Push services might need to limit the size and number of stored push messages to avoid overloading. To limit the size of messages, the push service MAY return the 413 (Payload Too Large) status code for messages that are too large. To limit the number of stored push messages, the push service MAY either expire messages prior to their advertised Time-To-Live or reduce their advertised Time-To-Live.

8.3. Subscription Expiration

In some cases, it may be necessary to terminate subscriptions so that they can be refreshed. This applies to both push message subscriptions and receipt subscriptions.

A push service can remove a subscription at any time. If a user agent or application server has an outstanding request to a subscription resource (see Section 7), this can be signaled by returning a 400-series status code, such as 410 (Gone).

A user agent or application server can request that a subscription be removed by sending a DELETE request to the push message subscription or receipt subscription URI.

A push service MUST return a 400-series status code, such as 404 (Not Found) or 410 (Gone) if an application server attempts to send a push message to a removed or expired push message subscription.

8.3.1. Subscription Set Expiration

A push service MAY expire a subscription set at any time which MUST also expire all push message subscriptions in the set. If a user agent has an outstanding request to a push subscription set (see Section 7.1) this can be signaled by returning a 400-series status code, such as 410 (Gone).

A user agent can request that a subscription set be removed by sending a DELETE request to the subscription set URI. This MUST also remove all push message subscriptions in the set.

If a specific push message subscription that is a member of a subscription set is expired or removed, then it MUST also be removed from its subscription set.

8.4. Implications for Application Reliability

A push service that does not support reliable delivery over intermittent network connections or failing applications on devices, forces the device to acknowledge receipt directly to the application server, incurring additional power drain in order to establish (usually secure) connections to the individual application servers.

Push message reliability can be important if messages contain information critical to the state of an application. Repairing state can be costly, particularly for devices with limited communications capacity. Knowing that a push message has been correctly received avoids costly retransmissions, polling and state resynchronization.

The availability of push message delivery receipts ensures that the application developer is not tempted to create alternative mechanisms for message delivery in case the push service fails to deliver a critical message. Setting up a polling mechanism or a backup messaging channel in order to compensate for these shortcomings negates almost all of the advantages a push service provides.

However, reliability might not be necessary for messages that are transient (e.g. an incoming call) or messages that are quickly superceded (e.g. the current number of unread emails).

8.5. Subscription Sets and Concurrent HTTP/2 streams

If the push service requires that the user agent use push message subscription sets, then it MAY limit the number of concurrently active streams with the SETTINGS_MAX_CONCURRENT_STREAMS parameter within a HTTP/2 SETTINGS frame [RFC7540]. The user agent MAY be limited to one concurrent stream to manage push message subscriptions and one concurrent stream for each subscription set returned by the push service. This could force the user agent to serialize subscription requests to the push service.

9. Security Considerations

This protocol MUST use HTTP over TLS [RFC2818]. This includes any communications between user agent and push service, plus communications between the application and the push service. All URIs therefore use the "https" scheme. This provides confidentiality and integrity protection for subscriptions and push messages from external parties.

9.1. Confidentiality from Push Service Access

The protection afforded by TLS does not protect content from the push service. Without additional safeguards, a push service 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 application server sending the push message and the application on the user agent that receives it are frequently just different instances of the same application, so no standardized protocol is needed to establish a proper security context. The process of providing the application server with subscription information provides a convenient medium for key agreement.

The Web Push API codifies this practice by requiring that each push subscription created by the browser be bound to a browser generated encryption key. Pushed messages are authenticated and decrypted by the browser before delivery to applications. This scheme ensures that the push service is unable to examine the contents of push messages.

The public key for a subscription ensures that applications using that subscription 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 service 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 URIs provided for push resources MUST NOT provide any basis to correlate communications for a given user agent. It MUST NOT be possible to correlate any two push resource URIs based solely on their contents. This allows a user agent to control correlation across different applications, or over time.

Similarly, the URIs provided by the push service to identify a push message MUST NOT provide any information that allows for correlation across subscriptions. Push message URIs for the same subscription MAY contain information that would allow correlation with the associated subscription or other push messages for that subscription.

User and device information MUST NOT be exposed through a push or push message URI.

In addition, push URIs established by the same user agent or push message URIs for the same subscription MUST NOT include any information that allows them to be correlated with the user agent.

Note: This need not be perfect as long as the resulting anonymity set (see [RFC6973], Section 6.1.1) is sufficiently large. A push URI necessarily identifies a push service or a single server instance. It is also possible that traffic analysis could be used to correlate subscriptions.

A user agent MUST be able to create new subscriptions with new identifiers at any time.

9.3. Authorization

This protocol does not define how a push service establishes whether a user agent is permitted to create a subscription, or whether push messages can be delivered to the user agent. A push service MAY choose to authorize requests based on any HTTP-compatible authorization method available, of which there are numerous options. The authorization process and any associated credentials are expected to be configured in the user agent along with the URI for the push service.

Authorization is managed using capability URLs for the push message subscription, push, and receipt subscription resources (see [CAP-URI]). A capability URL grants access to a resource based solely on knowledge of the URL.

Capability URLs are used for their "easy onward sharing" and "easy client API" properties. These make it possible to avoid relying on relationships between push services and application servers, with the protocols necessary to build and support those relationships.

Capability URLs act as bearer tokens. Knowledge of a push message subscription URI implies authorization to either receive push messages or delete the subscription. Knowledge of a push URI implies authorization to send push messages. Knowledge of a push message URI allows for reading and acknowledging that specific message. Knowledge of a receipt subscription URI implies authorization to receive push receipts. Knowledge of a receipt subscribe URI implies authorization to create subscriptions for receipts.

Note that the same receipt subscribe URI could be returned for multiple push message subscriptions. Using the same value for a large number of subscriptions allows application servers to reuse receipt subscriptions, which can provide a significant efficiency advantage. A push service that uses a common receipt subscribe URI loses control over the creation of receipt subscriptions. This can result in a potential exposure to denial of service; stateless resource creation can be used to mitigate the effects of this exposure.

Encoding a large amount of random entropy (at least 120 bits) in the path component ensures that it is difficult to successfully guess a valid capability URL.

9.4. Denial of Service Considerations

Discarding unwanted messages at the user agent based on message authentication doesn’t protect against a denial of service attack on the user agent. Even a relatively small volume of push messages can cause battery-powered devices to exhaust power reserves.

An application can limit where valid push messages can originate by limiting the distribution of push URIs to authorized entities. Ensuring that push URIs are hard to guess ensures that only application servers that have been given a push URI can use it.

A malicious application with a valid push URI could use the greater resources of a push service to mount a denial of service attack on a user agent. Push services SHOULD limit the rate at which push messages are sent to individual user agents. A push service or user agent MAY terminate subscriptions (Section 8.3) that receive too many push messages.

End-to-end confidentiality mechanisms, such as those in [API], prevent an entity with a valid push message subscription URI from learning the contents of push messages. Push messages that are not successfully authenticated will not be delivered by the API, but this can present a denial of service risk.

Conversely, a push service is also able to deny service to user agents. Intentional failure to deliver messages is difficult to distinguish from faults, which might occur due to transient network errors, interruptions in user agent availability, or genuine service outages.

9.5. Logging Risks

Server request logs can reveal subscription-related URIs. Acquiring a push message subscription URI enables the receipt of messages or deletion of the subscription. Acquiring a push URI permits the sending of push messages. Logging could also reveal relationships between different subscription-related URIs for the same user agent. Encrypted message contents are not revealed to the push service.

Limitations on log retention and strong access control mechanisms can ensure that URIs are not learned by unauthorized entities.

10. IANA Considerations

This protocol defines new HTTP header fields in Section 10.1. New link relation types are identified using the URNs defined in Section 10.2. Port registration is defined in Section 10.3

10.1. Header Field Registrations

HTTP header fields are registered within the "Message Headers" registry maintained at <https://www.iana.org/assignments/message-headers/>.

This document defines the following HTTP header fields, so their associated registry entries shall be added according to the permanent registrations below (see [RFC3864]):

Header Field Name	Protocol	Status	Reference
TTL	http	standard	Section 6.2
Push-Receipt	http	standard	Section 6.1

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

10.2. Link Relation URNs

This document registers URNs for use in identifying link relation types. These are added to a new "Web Push Identifiers" registry according to the procedures in Section 4 of [RFC3553]; the corresponding "push" sub-namespace is entered in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry.

The "Web Push Identifiers" registry operates under the IETF Review policy [RFC5226].

Registry name: Web Push Identifiers

URN Prefix: urn:ietf:params:push

Specification: (this document)

Repository: [Editor/IANA note: please include a link to the final registry location.]

Index value: Values in this registry are URNs or URN prefixes that start with the prefix "urn:ietf:params:push". Each is registered independently.

New registrations in the "Web Push Identifiers" are encouraged to include the following information:

URN: A complete URN or URN prefix.

Description: A summary description.

Specification: A reference to a specification describing the semantics of the URN or URN prefix.

Contact: Email for the person or group making the registration.

Index value: As described in [RFC3553], URN prefixes that are registered include a description of how the URN is constructed. This is not applicable for specific URNs.

These values are entered as the initial content of the "Web Push Identifiers" registry.

URN: urn:ietf:params:push

Description: This link relation type is used to identify a resource for sending push messages.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:set

Description: This link relation type is used to identify a collection of push message subscriptions.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:receipts

Description: This link relation type is used to identify a resource for creating new push message receipt subscriptions.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

10.3. Service Name and Port Number Registration

Service names and port numbers are registered within the "Service Name and Transport Protocol Port Number Registry" maintained at <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>.

IANA is requested to assign the System Port number 1001 and the service name "webpush" in accordance with [RFC6335].

Service Name.
webpush

Transport Protocol.
tcp

Assignee.
IESG (iesg@ietf.org)

Contact.
The Web Push WG (webpush@ietf.org)

Description.
HTTP Web Push

Reference.

[RFCthis]

Port Number.
1001

11. Acknowledgements

Significant technical input to this document has been provided by Ben Bangert, Kit Cambridge, JR Conlin, Matthew Kaufman, Costin Manolache, Mark Nottingham, Robert Sparks, Darshak Thakore and many others.

12. References

12.1. Normative References

[CAP-URI] Tennison, J., "Good Practices for Capability URLs", FPWD capability-urls, February 2014, <http://www.w3.org/TR/capability-urls/>.

[I-D.ietf-httpbis-alt-svc] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-09 (work in progress), May 2015.

[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.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, June 2003.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5382] Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", RFC 5382, October 2008.

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

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", RFC 6335, August 2011.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, April 2012.

[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.

[RFC7232] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

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

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2", RFC 7540, May 2015.

12.2. Informative References

[API] Sullivan, B., Fullea, E., and M. van Ouwerkerk, "Web Push API", ED push-api, February 2015, <https://w3c.github.io/push-api/>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, July 2013.

Appendix A. Change Log

[[The RFC Editor is requested to remove this section at publication.]]

A.1. Since draft-ietf-webpush-protocol-00

Editorial changes for Push Message Time-To-Live

Editorial changes for Push Acknowledgements

Removed subscription expiration based on HTTP cache headers

A.2. Since draft-ietf-webpush-protocol-01

Added Subscription Sets

Added System Port as an alternate service with guidance for idle timeouts

Finalized status codes for acknowledgements

Editorial changes for Rate Limits

Authors’ Addresses

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

Email: martin.thomson@gmail.com

Elio Damaggio
Microsoft
One Microsoft Way
Redmond, WA 98052
US

Email: elioda@microsoft.com

Brian Raymor (editor)
Microsoft
One Microsoft Way
Redmond, WA 98052
US

Email: brian.raymor@microsoft.com