Generic Event Delivery Using HTTP Push

Abstract

   This document describes a simple protocol for the delivery of real-
   time events to user agents. This scheme uses HTTP/2 server push.

Status of This Memo

   This is an Internet Standards Track document.

   This document is a product of the Internet Engineering Task Force
   (IETF). It represents the consensus of the IETF community. It has
   received public review and has been approved for publication by the
   Internet Engineering Steering Group (IESG). Further information on
   Internet Standards is available in Section 2 of RFC 7841.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc8030.

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

   o request the delivery of a push message to a user agent,

   o create new push message delivery subscriptions, and

   o monitor for new push messages.

   A standardized method of event delivery is particularly important for
   the W3C Push API, where application servers might need to use
   multiple push services. 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

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in [RFC2119].

   This document defines the following terms:

   application: Both the sender and the 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 usually
      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:

   o Subscribing for Push Messages (Section 4)

   o Requesting Push Message Delivery (Section 5)

   o Replacing Push Messages (Section 5.4)

   o Acknowledging Push Messages (Section 6.2)

When an example depends on HTTP/2 server push, the more verbose frame
format from [RFC7540] is used:

o Receiving Push Messages for a Subscription (Section 6)

o Receiving Push Messages for a Subscription Set (Section 6.1)

o Receiving Push Message Receipts (Section 6.3)

All examples use HTTPS over the default port (443) rather than the
registered port (1001). A push service deployment might prefer this
configuration to maximize chances for user agents to reach the
service. A push service might use HTTP alternative services to
redirect a user agent to the registered port (1001) to gain the
benefits of the standardized HTTPS port without sacrificing
reachability (see Section 3). This would only be apparent in the
examples through the inclusion of the Alt-Used header field [RFC7838]
in requests from the user agent to the push service.

Examples do not include specific methods for push message encryption
or application server authentication because the protocol does not
define a mandatory system. The examples in Voluntary Application
Server Identification [VAPID] and Message Encryption for WebPush
[ENCRYPT] demonstrate the approach adopted by the W3C Push API [API]
for its requirements.

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 |
   +-------+            +--------------+      |   Server    |
       |                       |              +-------------+
       |      Subscribe        |                    |
       |---------------------->|                    |
       |       Monitor         |                    |
       |<=====================>|                    |
       |                       |                    |
       |      Distribute Push Resource              |
       |------------------------------------------->|
       |                       |                    |
       :                       :                    :
       |                       |  Push Message      |
       |      Push Message     |<-------------------|
       |<--------------------- |                    |
       |                       |                    |
```
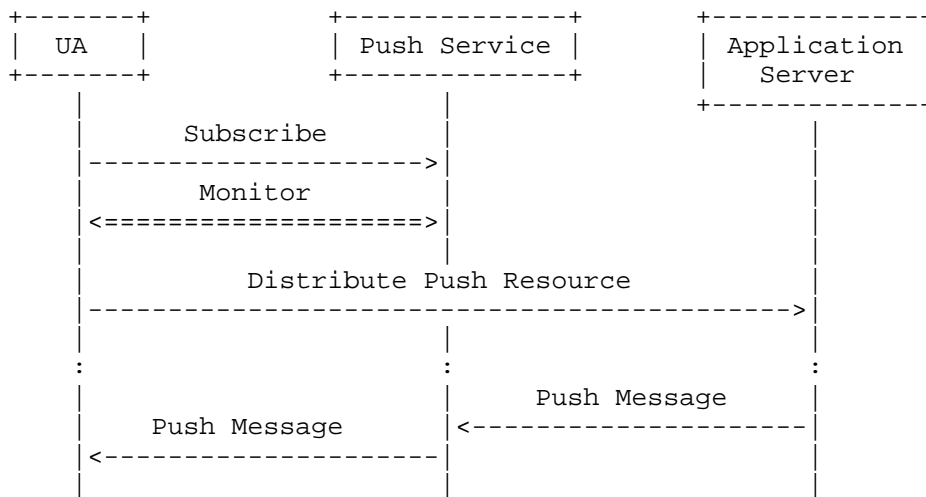
                   Figure 1: WebPush Architecture

   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. A subscription is used by the application server
   to send messages to the push service for delivery to the user agent.
   The user agent uses the subscription to monitor the push service for
   any incoming message.

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

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

   o 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 the user agent at any time. User
agents and application servers must be prepared to manage changes in
the 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 (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 6) 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 6.1) 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: An application server requests the delivery (Section 5) of a
   push message using a push resource. A link relation of type
   "urn:ietf:params:push" identifies a push resource.

push message: The push service creates a push message resource to
   identify push messages that have been accepted for delivery
   (Section 5). The push message resource is also deleted by the
   user agent to acknowledge receipt (Section 6.2) of a push message.

receipt subscription: An application server receives delivery
   confirmations (Section 5.1) for push messages using a receipt
   subscription. A link relation of type
   "urn:ietf:params:push:receipt" identifies a receipt subscription.

3. Connecting to the Push Service

   The push service MUST use HTTP over Transport Layer Security (TLS)
   [RFC2818] following the recommendations in [RFC7525]. 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 [RFC7838].

   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 states 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 201 (Created) response indicates that the push subscription was
   created. A URI for the push message subscription resource that was
   created in response to the request MUST be returned in the Location
   header field.

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

   An application-specific method is used to distribute the push URI to
   the application server. Confidentiality protection and application
   server authentication MUST be used to ensure that this URI is not
   disclosed to unauthorized recipients (Section 8.3).

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
      rel="urn:ietf:params:push"
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
      rel="urn:ietf:params:push:set"
Location: https://push.example.net/subscription/LBhhw0OohO-Wl4Oi971UG
```

4.1. Collecting Subscriptions into Sets

Collecting multiple push message subscriptions into a subscription
set can represent a significant efficiency improvement for push
services and user agents. The push service MAY provide a URI for a
subscription set resource in a link relation of type
"urn:ietf:params:push:set".

When a subscription set is returned in a push message subscription
response, the user agent SHOULD include this subscription set in a
link relation of type "urn:ietf:params:push:set" in subsequent
requests to create new push message subscriptions.

A user agent MAY omit the subscription set if it is unable to receive
push messages in an aggregated way for the lifetime of the
subscription. This might be necessary if the user agent is
monitoring subscriptions on behalf of other push message receivers.

```
POST /subscribe HTTP/1.1
Host: push.example.net
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
      rel="urn:ietf:params:push:set"
```

The push service SHOULD return the same subscription set in its
response, although it MAY return a new subscription set if it is
unable to reuse the one provided by the user agent.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </push/YBJNBIMwwA_Ag8EtD47J4A>;
      rel="urn:ietf:params:push"
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
      rel="urn:ietf:params:push:set"
Location: https://push.example.net/subscription/i-nQ3A9Zm4kgSWg8_ZijV
```

A push service MUST return a 400 (Bad Request) status code for
requests that contain an invalid subscription set. A push service
MAY return a 429 (Too Many Requests) status code [RFC6585] to reject
requests that omit a subscription set.

How a push service detects that requests originate from the same user
agent is implementation-specific but could take ambient information
into consideration, such as the TLS connection, source IP address,
and port. Implementers are reminded that some heuristics can produce
false positives and hence, cause requests to be rejected incorrectly.

5. Requesting Push Message Delivery

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

   POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
   Host: push.example.net
   TTL: 15
   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 MUST be returned 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/message/qDIYHNcfAIPP_5ITvURr-d6BGt

5.1. Requesting Push Message Receipts

   An application server can include the Prefer header field [RFC7240]
   with the "respond-async" preference to request confirmation from the
   push service when a push message is delivered and then acknowledged
   by the user agent. The push service MUST support delivery
   confirmations.

   POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
   Host: push.example.net
   Prefer: respond-async
   TTL: 15
   Content-Type: text/plain;charset=utf8
   Content-Length: 36

   iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB

When the push service accepts the message for delivery with
confirmation, it MUST return a 202 (Accepted) response. A URI for
the push message resource that was created in response to the request
MUST be returned in the Location header field. The push service MUST
also provide a URI for the receipt subscription resource in a link
relation of type "urn:ietf:params:push:receipt".

```
HTTP/1.1 202 Accepted
Date: Thu, 11 Dec 2014 23:56:55 GMT
Link: </receipt-subscription/3ZtI4YVNBnUUZhuoChl6omUvG4ZM>;
        rel="urn:ietf:params:push:receipt"
Location: https://push.example.net/message/qDIYHNcfAIPP_5ITvURr-d6BGt
```

For subsequent receipt requests to the same origin [RFC6454], the
application server SHOULD include the returned receipt subscription
in a link relation of type "urn:ietf:params:push:receipt". This
gives the push service the option to aggregate the receipts. The
push service SHOULD return the same receipt subscription in its
response, although it MAY return a new receipt subscription if it is
unable to reuse the one provided by the application server.

An application server MAY omit the receipt subscription if it is
unable to receive receipts in an aggregated way for the lifetime of
the receipt subscription. This might be necessary if the application
server is monitoring receipt subscriptions on behalf of the other
push message senders.

A push service MUST return a 400 (Bad Request) status code for
requests that contain an invalid receipt subscription. If a push
service wishes to limit the number of receipt subscriptions that it
maintains, it MAY return a 429 (Too Many Requests) status code
[RFC6585] to reject receipt requests that omit a receipt
subscription.

5.2. 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 MUST include the TTL (Time-To-Live) header
field in its request for push message delivery. The TTL header field
contains a value in seconds that suggests how long a push message is
retained by the push service.

The TTL rule specifies a non-negative integer, representing time in
seconds. A recipient parsing and converting a TTL value to binary
form SHOULD use an arithmetic type of at least 31 bits of non-
negative integer range. If a recipient receives a TTL value greater
than the greatest integer it can represent, or if any of its
subsequent calculations overflows, it MUST consider the value to be
2147483648 (2^31).

TTL = 1*DIGIT

A push service MUST return a 400 (Bad Request) status code in
response to requests that omit the TTL header field.

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

Once the 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.

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 an 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.

5.3. Push Message Urgency

   For a device that is battery-powered, it is often critical that it
   remains dormant for extended periods. Radio communication in
   particular consumes significant power and limits the length of time
   that the device can operate.

   To avoid consuming resources to receive trivial messages, it is
   helpful if an application server can communicate the urgency of a
   message and if the user agent can request that the push server only
   forwards messages of a specific urgency.

   An application server MAY include an Urgency header field in its
   request for push message delivery. This header field indicates the
   message urgency. The push service MUST NOT forward the Urgency
   header field to the user agent. A push message without the Urgency
   header field defaults to a value of "normal".

   A user agent MAY include the Urgency header field when monitoring for
   push messages to indicate the lowest urgency of push messages that it
   is willing to receive. A push service MUST NOT deliver push messages
   with lower urgency than the value indicated by the user agent in its
   monitoring request. Push messages of any urgency are delivered to a
   user agent that does not include an Urgency header field when
   monitoring for messages.

   The grammar for the Urgency header field is as follows:

   Urgency = urgency-option
   urgency-option = ("very-low" / "low" / "normal" / "high")

   In order of increasing urgency:

   +----------+----------------------------+--------------------------+
   | Urgency | Device State | Example Application |
   | | | Scenario |
   +----------+----------------------------+--------------------------+
very-low	On power and Wi-Fi	Advertisements
low	On either power or Wi-Fi	Topic updates
normal	On neither power nor Wi-Fi	Chat or Calendar Message
high	Low battery	Incoming phone call or
		time-sensitive alert
   +----------+----------------------------+--------------------------+

                    Table 1: Illustrative Urgency Values

   Multiple values for the Urgency header field MUST NOT be included in
   requests; otherwise, the push service MUST return a 400 (Bad Request)
   status code.

5.4. Replacing Push Messages

   A push message that has been stored by the push service can be
   replaced with new content. If the user agent is offline during the
   time that the push messages are sent, updating a push message avoids
   the situation where outdated or redundant messages are sent to the
   user agent.

   Only push messages that have been assigned a topic can be replaced.
   A push message with a topic replaces any outstanding push message
   with an identical topic.

   A push message topic is a string carried in a Topic header field. A
   topic is used to correlate push messages sent to the same
   subscription and does not convey any other semantics.

   The grammar for the Topic header field uses the "token" rule defined
   in [RFC7230].

   Topic = token

   For use with this protocol, the Topic header field MUST be restricted
   to no more than 32 characters from the URL and a filename-safe Base
   64 alphabet [RFC4648]. A push service that receives a request with a
   Topic header field that does not meet these constraints MUST return a
   400 (Bad Request) status code to the application server.

   A push message replacement request creates a new push message
   resource and simultaneously deletes any existing message resource
   that has a matching topic. If an attempt was made to deliver the
   deleted push message, an acknowledgment could arrive at the push
   service after the push message has been replaced. Delivery receipts
   for such deleted messages SHOULD be suppressed.

   The replacement request also replaces the stored TTL, Urgency, and
   any receipt subscription associated with the previous message in the
   matching topic.

   A push message with a topic that is not shared by an outstanding
   message to the same subscription is stored or delivered as normal.

For example, the following message could cause an existing message to
be replaced:

```
POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
TTL: 600
Topic: upd
Content-Type: text/plain;charset=utf8
Content-Length: 36

ZuHSZPKa2b1jtOKLGpWrcrn8cNqt0iVQyroF
```

If the push service identifies an outstanding push message with a
topic of "upd", then that message resource is deleted. A 201
(Created) response indicates that the push message replacement was
accepted. A URI for the new push message resource that was created
in response to the request is included in the Location header field.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:57:02 GMT
Location: https://push.example.net/message/qDIYHNcfAIPP_5ITvURr-d6BGt
```

The value of the Topic header field MUST NOT be forwarded to user
agents. Its value is neither encrypted nor authenticated.

6. 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; instead, it uses HTTP/2
   server push [RFC7540] to send the contents of push messages as they
   are sent by application servers.

   A user agent MAY include an Urgency header field in its request. The
   push service MUST NOT deliver messages with lower urgency than the
   value of the header field as defined in Table 1 (Illustrative Urgency
   Values).

   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 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         = /subscription/LBhhw0OohO-Wl4Oi971UG
  :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
generated in association with the initial request. The response for
the server push includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method       = GET
  :path         = /message/qDIYHNcfAIPP_5ITvURr-d6BGt
  :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          = </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                    rel="urn:ietf:params:push"
  content-type  = text/plain;charset=utf8
  content-length = 36

DATA          [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB

HEADERS       [stream 7] +END_STREAM +END_HEADERS
  :status       = 200
```

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" preference 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.

6.1. Receiving Push Messages for a Subscription Set

   There are minor differences between receiving push messages for a
   subscription and a subscription set. If a subscription set is
   available, the user agent SHOULD use the subscription set to monitor
   for push messages rather than individual push message subscriptions.

   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; instead, it uses HTTP/2 server push
   [RFC7540] to send the contents of push messages as they are sent by
   application servers.

   A user agent MAY include an Urgency header field in its request. The
   push service MUST NOT deliver messages with lower urgency than the
   value of the header field as defined in Table 1 (Illustrative Urgency
   Values).

   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   = /subscription-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
   generated in association with the initial request. The server push's
   response includes the push message.

   PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
     :method   = GET
     :path   = /message/qDIYHNcfAIPP_5ITvURr-d6BGt
     :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
  link           = </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                   rel="urn:ietf:params:push"
  cache-control  = private
  content-type   = text/plain;charset=utf8
  content-length = 36

DATA           [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB

HEADERS        [stream 7] +END_STREAM +END_HEADERS
  :status        = 200
```

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" preference 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.

6.2. 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 an HTTP DELETE on the push message resource.

```
DELETE /message/qDIYHNcfAIPP_5ITvURr-d6BGt 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 return a 204
   (No Content) response to the application server monitoring the
   receipt subscription.

   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 return a 410
(Gone) response to the application server monitoring the receipt
subscription.

6.3. Receiving Push Message Receipts

The application server requests the delivery of receipts from the
push service by making an HTTP GET request to the receipt
subscription resource. The push service does not respond to this
request; instead, it uses HTTP/2 server push [RFC7540] to send push
receipts when messages are acknowledged (Section 6.2) 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. The response includes
a status code indicating the result of the message delivery and
carries no data.

The following example request is made over HTTP/2.

```
HEADERS       [stream 13] +END_STREAM +END_HEADERS
  :method       = GET
  :path         = /receipt-subscription/3ZtI4YVNBnUUZhuoChl6omUvG4ZM
  :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         = /message/qDIYHNcfAIPP_5ITvURr-d6BGt
  :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).

7. Operational Considerations

7.1. 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 HTTP
   alternative services [RFC7838]. HTTP alternative services allows for
   redistribution of load while maintaining the same URIs for various
   resources. A user agent can ensure a graceful transition by using
   the GOAWAY frame once it has established a replacement connection.

7.2. 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 (Section 5.2).

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

   A GET request to a push message subscription resource with only
   expired messages results in a response as though no push message was
   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 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 4096 bytes or less in size.

   To limit the number of stored push messages, the push service MAY
   respond with a shorter Time-To-Live than proposed by the application
   server in its request for push message delivery (Section 5.2). Once
   a message has been accepted, the push service MAY later expire the
   message prior to its advertised Time-To-Live. If the application
   server requested a delivery receipt, the push service MUST return a
   failure response (Section 6.2).

7.3. 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 MAY expire a subscription at any time. If there are
   outstanding requests to an expired push message subscription resource
   (Section 6) from a user agent or to an expired receipt subscription
   resource (Section 6.3) from an application server, this MUST be
   signaled by returning a 404 (Not Found) status code.

   A push service MUST return a 404 (Not Found) status code if an
   application server attempts to send a push message to an expired push
   message subscription.

   A user agent can remove its push message subscription by sending a
   DELETE request to the corresponding URI. An application server can
   remove its receipt subscription by sending a DELETE request to the
   corresponding URI.

7.3.1. Subscription Set Expiration

   A push service MAY expire a subscription set at any time and MUST
   also expire all push message subscriptions in the set. If a user
   agent has an outstanding request to a push subscription set
   (Section 6.1), this MUST be signaled by returning a 404 (Not Found)
   status code.

   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.

7.4. 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 and
   maintain (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 the
   state can be expensive, particularly for devices with limited
   communications capacity. Knowing that a push message has been
   correctly received avoids 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
   superseded (e.g., the current number of unread emails).

7.5. 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 an 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.

8. Security Considerations

   This protocol MUST use HTTP over TLS [RFC2818] following the
   recommendations in [RFC7525]. This includes any communications
   between the user agent and the push service, plus communications
   between the application server 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.

8.1. Confidentiality from Push Service Access

   The protection afforded by TLS does not protect content from the push
   service. Without additional safeguards, a push service can inspect
   and modify the message content.

   Applications using this protocol MUST use mechanisms that provide
   end-to-end confidentiality, integrity, and data origin
   authentication. 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
   distribution of subscription information from the user agent to its
   application server also offers a convenient medium for key agreement.

   For this requirement, the W3C Push API [API] has adopted Message
   Encryption for WebPush [ENCRYPT] to secure the content of messages
   from the push service. Other scenarios can be addressed by similar
   policies.

   The Topic header field exposes information that allows more granular
   correlation of push messages on the same subject. This might be used
   to aid traffic analysis of push messages by the push service.

8.2. 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. Of course, this does not
   prevent correlation using other information that a user agent might
   expose.

   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 ([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.

8.3. 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 multiple options
(including experimental options) with varying levels of security.
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 ([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 properties make it possible to avoid
relying on prearranged relationships or additional protocols between
push services and application servers.

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.

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.

8.4. Denial-of-Service Considerations

   A user agent can control where valid push messages originate by
   limiting the distribution of push URIs to authorized application
   servers. Ensuring that push URIs are hard to guess ensures that only
   application servers that have received a push URI can use it.

   Push messages that are not successfully authenticated by the user
   agent will not be delivered, but this can present a denial-of-service
   risk. Even a relatively small volume of push messages can cause
   battery-powered devices to exhaust power reserves.

   To address this case, the W3C Push API [API] has adopted Voluntary
   Application Server Identification [VAPID], which allows a user agent
   to restrict a subscription to a specific application server. The
   push service can then identify and reject unwanted messages without
   contacting the user agent.

   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 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 or user agent MAY also terminate subscriptions
   (Section 7.3) that receive too many push messages.

   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.

8.5. Logging Risks

   Server request logs can reveal subscription-related URIs or
   relationships between subscription-related URIs for the same user
   agent. Limitations on log retention and strong access control
   mechanisms can ensure that URIs are not revealed to unauthorized
   entities.

9. IANA Considerations

   This protocol defines new HTTP header fields in Section 9.1. New
   link relation types are identified using the URNs defined in
   Section 9.2. Port registration is defined in Section 9.3

9.1. 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, and the
   following entries have been added to the "Permanent Message Header
   Field Names" registry ([RFC3864]):

   +-------------------+----------+----------+--------------+
   | Header Field Name | Protocol | Status | Reference |
   +-------------------+----------+----------+--------------+
TTL	http	standard	Section 5.2
Urgency	http	standard	Section 5.3
Topic	http	standard	Section 5.4
   +-------------------+----------+----------+--------------+

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

9.2. Link Relation URNs

   This document registers URNs for use in identifying link relation
   types. These have been added to a new "Web Push Identifiers"
   registry according to the procedures in Section 4 of [RFC3553]; the
   corresponding "push" sub-namespace has been 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: RFC 8030 (this document)

   Repository: http://www.iana.org/assignments/webpush-parameters/

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

Registrations in the "Web Push Identifiers" registry include the
following information:

URN: A complete URN or URN prefix.

Description: A summary description.

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

Index Value: As described in [RFC3553]

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

   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.

Contact: The WEBPUSH WG of the IETF (webpush@ietf.org)

Reference: RFC 8030 (this document)

URN: urn:ietf:params:push:set

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

Contact: The WEBPUSH WG of the IETF (webpush@ietf.org)

Reference: RFC 8030 (this document)

URN: urn:ietf:params:push:receipt

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

Contact: The WEBPUSH WG of the IETF (webpush@ietf.org)

      Reference: RFC 8030 (this document)

9.3. 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/>.

   In accordance with [RFC6335], IANA has assigned the System Port
   number 1001 and the service name "webpush".

   Service Name:
      webpush

   Port Number:
      1001

   Transport Protocol:
      tcp

   Description:
      HTTP Web Push

   Assignee:
      The IESG (iesg@ietf.org)

   Contact:
      The IETF Chair (chair@ietf.org)

   Reference:
      RFC 8030 (this document)

10. References

10.1. Normative References

   [CAP-URI] Tennison, J., "Good Practices for Capability URLs", W3C
              First Public Working Draft capability-urls, February 2014,
              <http://www.w3.org/TR/capability-urls/>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <http://www.rfc-editor.org/info/rfc2119>.

   [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818,
              DOI 10.17487/RFC2818, May 2000,
              <http://www.rfc-editor.org/info/rfc2818>.

   [RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An
              IETF URN Sub-namespace for Registered Protocol
              Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June
              2003, <http://www.rfc-editor.org/info/rfc3553>.

   [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration
              Procedures for Message Header Fields", BCP 90, RFC 3864,
              DOI 10.17487/RFC3864, September 2004,
              <http://www.rfc-editor.org/info/rfc3864>.

   [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data
              Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006,
              <http://www.rfc-editor.org/info/rfc4648>.

   [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
              IANA Considerations Section in RFCs", BCP 26, RFC 5226,
              DOI 10.17487/RFC5226, May 2008,
              <http://www.rfc-editor.org/info/rfc5226>.

   [RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P.
              Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142,
              RFC 5382, DOI 10.17487/RFC5382, October 2008,
              <http://www.rfc-editor.org/info/rfc5382>.

   [RFC5988] Nottingham, M., "Web Linking", RFC 5988,
              DOI 10.17487/RFC5988, October 2010,
              <http://www.rfc-editor.org/info/rfc5988>.

   [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", BCP 165,
              RFC 6335, DOI 10.17487/RFC6335, August 2011,
              <http://www.rfc-editor.org/info/rfc6335>.

   [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454,
              DOI 10.17487/RFC6454, December 2011,
              <http://www.rfc-editor.org/info/rfc6454>.

   [RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status
              Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012,
              <http://www.rfc-editor.org/info/rfc6585>.

   [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <http://www.rfc-editor.org/info/rfc7230>.

   [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Semantics and Content", RFC 7231,
              DOI 10.17487/RFC7231, June 2014,
              <http://www.rfc-editor.org/info/rfc7231>.

   [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Conditional Requests", RFC 7232,
              DOI 10.17487/RFC7232, June 2014,
              <http://www.rfc-editor.org/info/rfc7232>.

   [RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240,
              DOI 10.17487/RFC7240, June 2014,
              <http://www.rfc-editor.org/info/rfc7240>.

   [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre,
              "Recommendations for Secure Use of Transport Layer
              Security (TLS) and Datagram Transport Layer Security
              (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May
              2015, <http://www.rfc-editor.org/info/rfc7525>.

   [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext
              Transfer Protocol Version 2 (HTTP/2)", RFC 7540,
              DOI 10.17487/RFC7540, May 2015,
              <http://www.rfc-editor.org/info/rfc7540>.

   [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP
              Alternative Services", RFC 7838, DOI 10.17487/RFC7838,
              April 2016, <http://www.rfc-editor.org/info/rfc7838>.

10.2. Informative References

   [API]   Beverloo, P., Thomson, M., van Ouwerkerk, M., Sullivan,
              B., and E. Fullea, "Push API", W3C Editor's Draft push-
              api, November 2016, <https://w3c.github.io/push-api/>.

   [ENCRYPT] Thomson, M., "Message Encryption for Web Push", Work in
              Progress, draft-ietf-webpush-encryption-06, October 2016.

   [RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J.,
              Morris, J., Hansen, M., and R. Smith, "Privacy
              Considerations for Internet Protocols", RFC 6973,
              DOI 10.17487/RFC6973, July 2013,
              <http://www.rfc-editor.org/info/rfc6973>.

   [VAPID]   Thomson, M. and P. Beverloo, "Voluntary Application Server
              Identification for Web Push", Work in Progress,
              draft-ietf-webpush-vapid-01, June 2016.

Authors' Addresses

   Martin Thomson
   Mozilla
   331 E Evelyn Street
   Mountain View, CA 94041
   United States of America

   Email: martin.thomson@gmail.com

   Elio Damaggio
   Microsoft
   One Microsoft Way
   Redmond, WA 98052
   United States of America

   Email: elioda@microsoft.com

   Brian Raymor (editor)
   Microsoft
   One Microsoft Way
   Redmond, WA 98052
   United States of America

   Email: brian.raymor@microsoft.com