Realization of Per-Resource Server Push Using RFC 8030 and Push API

Yuto Ito
Faculty of Informatics
Kogakuin University
Tokyo, Japan
em20004@ns.kogakuin.ac.jp

Yoshifumi Manabe
Faculty of Informatics
Kogakuin University
Tokyo, Japan
manabe@cc.kogakuin.ac.jp

***Abstract*—Various web services use a technology that sends messages from the server to the client asynchronously without the user's operation. However, web service companies and OSSs use different mechanisms to realize the technology. We define the requirements for this mechanism, survey related technologies, and devise a generic architecture to realize this mechanism with using open technologies as much as possible. As a result, the requirements can be met by using the message delivery mechanism of RFC 8030 as a basis and adding original extensions such as message deduplication and pub/sub.**

***Index Terms*—Server Push, Web Push, Push Notification, Pub/Sub Messaging, Exactly-once delivery**

I. Introduction

A. Server Push

In Web services, there are two patterns in which the server passes the processing results to a client. The first is a general pattern in which the server that received an HTTP request sent by a client executes all the processing synchronously and returns the results as a response, as shown in Fig.1(a).

The second is a pattern with asynchronous processing as shown in Fig.1 (b). The server that receives an HTTP request from a client executes some processing and returns a response. The server executes the rest of the processing asynchronously with the request and sends the result to the client in some way. Since this pattern seems to push messages from the server, it is called a server push [1]–[4].

In a simple Web service, all data can be exchanged by a request and a response in the first pattern. However, web services that provide an excellent UX may update the information to be displayed asynchronously with the user's operations. For example, SNS services display messages newly posted by other users without the user's operation. Services that take a long time to perform processing such as file compression/decompression and file format conversion may get processing results asynchronously from the viewpoint of UX and implementations.

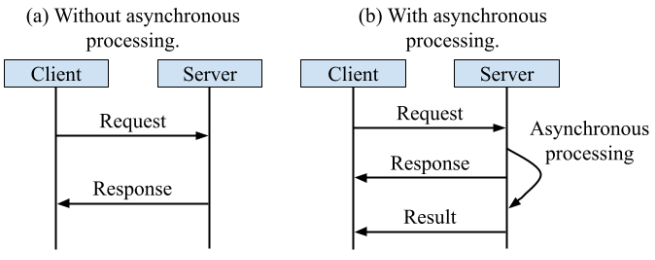

Fig. 1. Web service communication flow.

B. Per-Resource Server Push

There are various technologies and architectures to realize a server push. The implementation methods vary from service to service. RFC 8030 - Generic Event Delivery Using HTTP Push [5] (hereafter referred to as RFC 8030) and the Push API [6] provide a standard server push mechanism in web browsers. This mechanism builds a general-purpose server push route between a user's Web browser and the Web server. Web services can freely send messages to the browser using this route. For example, a news site wants to send a desktop notification to all users when new news is posted. It can use this mechanism to send a message to all users. However, when the information displayed on the screen that a user is currently viewing an SNS is updated in the server, there is no mechanism to deliver the new information by server push only to the users who need it. In this paper, the mechanism for realizing such a request is called a per-resource server push.

C. Contribution of This Paper

When a server push is required for a Web service, each company is currently developing its mechanism to realize it. Though there are some Open Source Softwares such as Socket.IO [7], [8], Gotify [9], Plasma [10] that can easily realize a server push, the mechanisms are not united. As mentioned above, RFC 8030 and Push API provide a general-purpose mechanism for realizing a server push in the browser, however there is no standard mechanism for realizing a per-resource server push.

First, this paper summarizes the requirements necessary to realize a general-purpose per-resource server push. Next, we investigate the technologies and existing works required to meet the requirements. Finally, we propose a new mechanism

978-1-6654-3610-6/21/$31.00 ©2021 IEEE

that combines them to realize a general-purpose per-resource server push. A server push is also used in mobile devices other than Web browsers. Since mobile devices often have a mechanism unique to each vendor, this research targets only Web browsers.

II. Requirements of Per-Resource Server Push

This section clarifies the requirements of the per-resource server push and describes the reason of the each requirement.

The following four requirements can be considered to realize the per-resource server push.

1) Asynchronous message delivery
2) Exactly-once message delivery
3) Scale-out design
4) Mechanism like pub/sub

The first three are the requirements for a simple server push, and the fourth is a unique requirement for the per-resource server push.

A. Asynchronous Message Delivery

The value of asynchronous message delivery has already been explained in Section 1 as the demand for a server push, so it is omitted here.

B. Exactly-Once Message Delivery

The client does not want to process the same message over and over again. For example, if the received message is displayed as it is as a desktop notification, the same message will be displayed many times. Also, if the client sends a request to the server based on the message, it is wasteful that the client sends many requests. Even more problematic is the case where the message is lost in transit for some reason and the client cannot receive the message. Therefore, it is necessary to have a mechanism in which the message is always delivered and the client receives the message only once without duplication.

C. Scale-Out Design

Scalability is very important for Web services, so the server is usually designed to be able to be scaled out. Similarly, the per-resource server push mechanism needs to be designed to scale out.

D. Mechanism Like Pub/Sub

In the per-resource server push, when the state of the resource existing on the server changes, it is necessary to notify the clients that need it. For that purpose, it is necessary to have a mechanism to manage which client needs which resource state transition information, and a mechanism to communicate this information to the necessary clients when a resource state transition occurs. For example, consider a situation in which clients A and B want to receive a message when there is a change in the inventory quantity of product X on an EC site. At this time, A and B tell the server to send a message if there is a change in X. This process is called a subscribe. If there is a change in X, the server will send a message to all clients currently subscribing to X. This makes it possible to deliver a message to the clients who need the message. Such a mechanism is generally called pub/sub [11], [12].

III. Server Push Related Technology

We introduce the following six technologies that may be used to realize a server push.

1) Polling
2) Long polling [13]
3) Server-sent events [14]
4) WebSocket [15]–[17]
5) Socket.IO [7], [8]
6) RFC 8030 [5]

A. Polling

As shown in Fig.2(a), polling is a technique that continues sending messages to the server intermittently to check for new messages. Polling has the feature that a response is returned immediately with or without a message.

B. Long Polling

Long polling is a technique similar to polling as shown in Fig.2(b). Unlike polling, long polling has the feature that the server does not return a response and keeps the session open until a new message occurs.

C. Server-Sent Events

Server-sent events is a mechanism that enables streaming communication via HTTP as shown in Fig.2(c). Polling and long polling can return only one response to a request. Server-sent events do not close the session after returning a response, so multiple responses can be returned. Server-sent events have higher real-time performance and less overhead than long polling.

D. WebSocket

WebSocket is a protocol that provides bidirectional communication and is located on the same layer as HTTP. Unlike HTTP, WebSocket is made on the premise of bidirectional communication, so it is relatively easy to send messages from the server. WebSocket continues to use the same session once established like Server-sent events.

E. Socket.IO

Socket.IO is a mechanism that provides a message delivery called pub/sub or event-driven messaging. Socket.IO is one layer above HTTP and WebSocket. When a message is added to an event, Socket.IO delivers the message to all clients listening for the event via WebSocket or long polling.

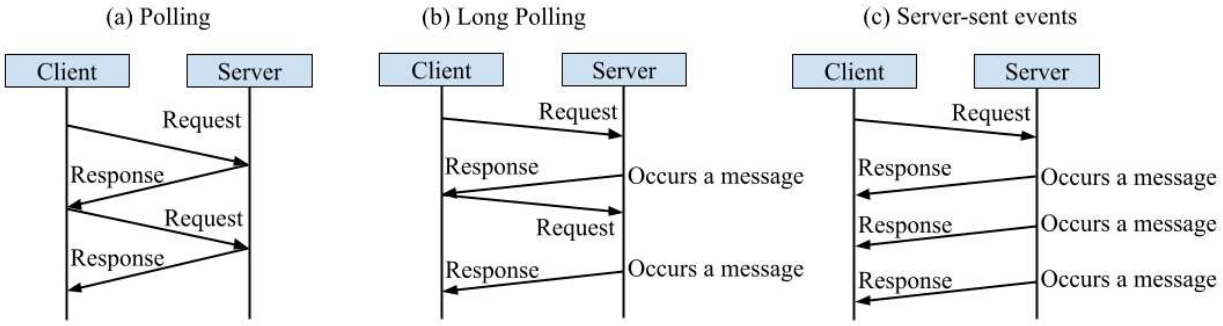

Fig. 2. Process flow of each technology.

F. RFC 8030

RFC 8030 is a general-purpose server push mechanism that can be used in Web browsers and mobile terminals. RFC 8030 defines the behavior required for a server push, such as creating, managing, and deleting messages and delivery routes. Since RFC 8030 is expected to be also used on mobile devices, it includes specifications for reducing power consumption and communication capacity. However, since mobile devices are not covered in this study, these explanations are omitted. The JavaScript API and Service Worker [18] extensions required to use RFC 8030 from a Web browser are defined in the Push API.

The features of RFC 8030 are that (1) it provides an independent service that is responsible for retaining and delivering messages, and (2) it retains messages without deleting them until the delivery of the message is confirmed. Fig.3 shows the general flow of a server push using RFC 8030 and Push API.

The outline of the terms shown in Fig.3 is described below.

- Web Page: JavaScript running on the Web Page that a user is viewing. It operates independently for each tab.
- Service Worker: A JavaScript worker process that is shared between tabs and runs in the background. Although it has various uses, it has the role of passing the messages passed by the User Agent to the Web Page in the Push API.
- User Agent: Web browser
- Push Service: A service responsible for managing Push Subscriptions and storing and delivering messages.
- Application Server: Web service's API server.
- Push Subscription: Communication route identifier for sending messages.

The processing flow of RFC 8030 and Push API is as follows.

1) A Web Page requests the Push Service to create a Push Subscription via the User Agent. The Web Page gets the created Push Subscription identifier as a return value.
2) The Web Page registers the Push Subscription to the Application Server.
3) When the Application Server wants to send a message to the Web Page, The Application Server sends the message to the Push Service using the Push Subscription.
4) The Push Service sends the message to the User Agent.
5) The User Agent invokes a PushEvent to convey the received message to the Service Worker. PushEvent is an event that sends data from the User Agent to the Service Worker.
6) The Service Worker sends the received message to the Web Page with MessageEvent as needed. MessageEvent is an event that sends data from the Service Worker to the Web Page.
7) When all PushEvent invokes are complete, the User Agent informs the Push Service that the message has been received. Then the Push Service deletes the message.

IV. Proposed Method

In this section, we first consider what is necessary to meet each requirement mentioned in Section 2. Next, we select appropriate technologies and devise solutions for each requirement. Finally, we summarize the architecture for realizing the per-resource server push.

A. Solution for Each Requirement

1) Asynchronous Message Delivery: This requirement is so simple that any of the technologies introduced in Section 3 can achieve it. Technologies other than Socket.IO and RFC 8030 are just setting single message delivery route, then there is no mechanism to distribute messages generated in a server to appropriate delivery routes. Therefore, if we use technologies other than Socket.IO and RFC 8030, we need to develop this mechanism.

2) Exactly-Once Message Delivery: There are some cases to consider when implementing an exactly-once communication over the network. It is the case where a message is lost on the network in transit, or a server and a client are temporarily disconnected and messages cannot be delivered.

For example, even with technologies such as Server-sent events, WebSocket, and Socket.IO that keep a session, the session may be disconnected for some reason. Also, in the case of a rolling update of a server, a session is forcibly

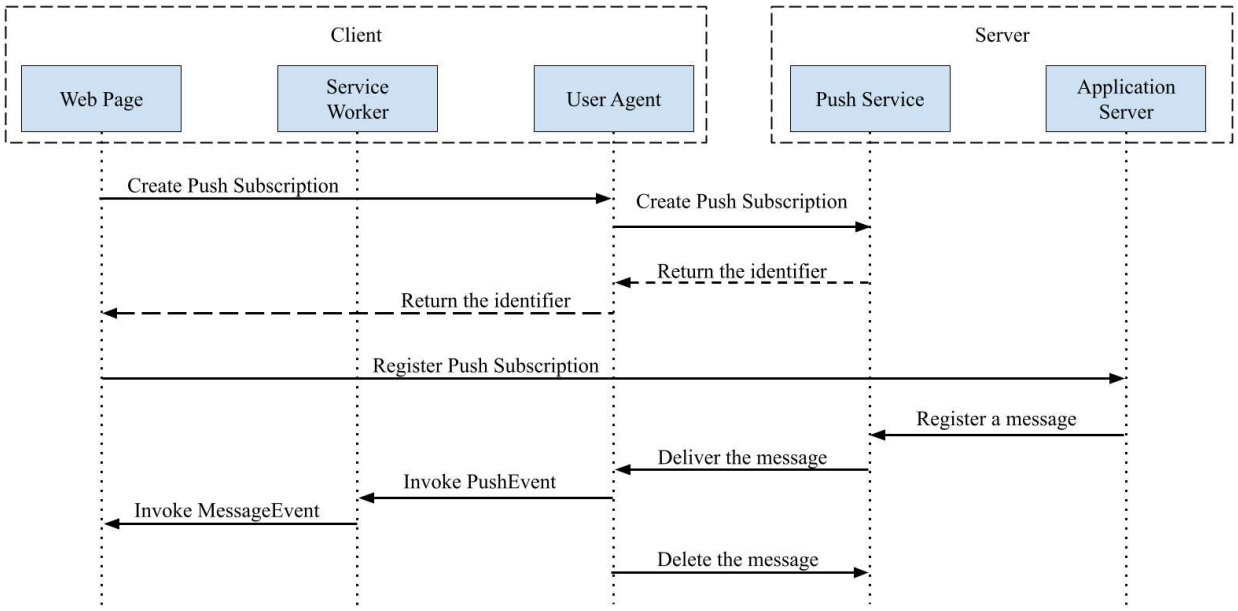

Fig. 3. Communication flow of RFC 8030 and Push API.

disconnected. At this time, if no measures are taken, the messages generated during the rolling update are lost. Besides the above cases, communication over the network can fail at any time. So a mechanism is required for a server to resend the message when a client fails to receive a message.

A server alone cannot determine whether a client has received a message. Therefore, in order to realize an exactly-once message delivery, a mechanism such as a read management is required to notify the server that a client has received a message. If a server resends a message, a client can receive the message, however the message may be duplicated.

It is difficult to prevent message duplication with a server alone. Therefore, a server guarantees delivery at least once and each client removes duplication. For deduplication, a server assigns an identifier to each message and a client retains the identifier of the received messages. The client ignores the newly received message if the message's identifier is known. This allows a pseudo exactly once message delivery. If a client wants to completely deduplication, the client needs to retain all identifiers, however it puts pressure on a storage capacity. In reality, the client will either limit the number of identifiers or set an expiration time.

A server cannot delete a message until a client receives the message, so if the client disconnect process is not performed correctly, the number of messages will continue to increase on the server. In order to solve this problem, a mechanism is required to set an expiration time for each message and delete expired messages on the server.

The elements necessary to achieve this requirement are summarized below.

(1) Message management (retention, read management, resend, expiration time)
(2) Message deduplication on the client
(3) Mapping a client and a message

Among the technologies introduced in Section 3, RFC 8030 supports (1) and (3), so the cost of achieving this requirement is the lowest. RFC 8030 does not support (2), however it is relatively easy to implement.

3) Scale-Out Design: Server-sent event, WebSocket, Socket.IO continue to use an established session. However, when the per-resource server push scale out a message delivery server, the per-resource server push needs to disconnect and reconnect a session to rebalance. A mechanism for managing messages as described in 4.A.2 is required so that messages are not lost during a session disconnection. Therefore, RFC 8030 is still the best choice for this requirement.

In addition, the per-resource server push needs to be able to scale out the storage that retains messages. In RFC 8030, the Push Service retains messages. Currently, the Push Service is fixed for each browser, and each browser vendor develops and operates this service. Therefore, the scalability of message delivery is not considered in this paper.

4) Mechanism Like Pub/Sub: In the technology introduced in Section 3, only Socket.IO supports pub/sub. If we just want to support pub/sub, using Socket.IO is the lowest cost solution. However, since Socket.IO does not have a message management mechanism, the cost of realizing the requirements of 4.A.2 and 4.A.3 is high. Therefore, this requirement is considered to be based on RFC 8030, which has a low implementation cost for requirements of 4.A.2 and 4.A.3.

In order to realize a pub/sub on RFC 8030, it is necessary to have a mechanism in which a server retains a resource-client mapping and a client updates the mapping. The client specifically means the tab displaying a web page. The per-resource server push needs to be able to pub/sub resources

for each tab. A tab needs to unsubscribe its resources when it is closed. However, the termination process is not always executed normally, then the subscription is not terminated. As a countermeasure, a server sets an expiration time for each tab-resource mapping. A client needs to update the expiration time at regular intervals.

Since multiple tabs subscribe to multiple resources in one Web Page, the Application Server needs to distinguish between them when delivering messages. However, the Push API has a limit of one Push Subscription per Service Worker. Basically, one Service Worker is created in one origin (origin is a combination of URL scheme, host, and port) and shared between tabs. Therefore, one Push Subscription is shared by multiple tabs and multiple resources. To be precise, a Service Worker is linked to one origin and an URL path, so we can create multiple Service Workers and Push Subscriptions in the same origin. However, it is not realistic because the life cycle of a Service Workers becomes complicated.

When a Service Worker receives a message, it sends the message to all related tabs. Therefore, if one Push Subscription is shared by multiple tabs and multiple resources, a tab will receive the other tab's messages. The server needs to include a tab identifier and a resource identifier in each message so that each tab can only process messages it needs. A resource identifier is prepared independently by each service. For example, a combination of a relational database's table name and its primary key can be used as a resource identifier. A tab identifier is generated by each tab. A tab also sends the tab's identifier to the server when it changes the subscribed resources. This makes it possible to build a mechanism for independently resource subscribing and message receiving for each tab.

B. Architecture

The elements required to realize each requirement are summarized as follows. The unique elements of this paper are underlined.

1) Asynchronous message delivery
2) Message management (retention, read management, re-send, expiration time)
3) Mapping a client and a message
4) Message management server scalability
5) Exactly-once message delivery
6) pub/sub resources for each tab

Among the technologies introduced in Section 3, RFC 8030 covers most of the above elements. In this paper, we select RFC 8030 as the base technology for the per-resource server push.

The role of RFC 8030 on the per-resource server push is to deliver messages from an Application Server to a User Agent asynchronously at least once. The per-resource server push requires exactly-once, however RFC 8030 can only guarantee at-least-once. As a solution, an Application Server assigns an identifier to each message as described in 4.A.2, and a Web Page performs deduplication based on the message identifier. This makes it possible to realize a pseudo exactly-once on RFC 8030.

RFC 8030 cannot determine which tab subscribes to which resource. In order to pub/sub resources for each tab, it is necessary to assign an identifier to each tab and resource as described in 4.A.4 and build a mechanism to realize pub/sub using assigned identifiers.

The data structures and processing flows based on the requirements are described below. The unique elements of this paper are underlined.

Message format for delivery.

- Message identifier
- Tab identifier
- Resource identifier
- Arbitrary data(Body)

Mapping of Web Page and resource retained by Application Server.

- Tab identifier
- Push Subscription identifier
- Resource identifier
- Expiration time

Resource subscription flow.

1) Web Page prepares a Push Subscription and registers to an Application Server
2) Web Page generates a tab identifier
3) Web Page updates subscription resources
   - Web Page sends the tab identifier together
4) After that, Web Page updates subscription resources as needed
   - To renew an expiration time
   - To change subscription resources

Message sending and receiving flow.

1) Application Server sends a message to a Push Service
2) Service Worker receives the message from the Push Service and delivers the message to a Web Page with MessageEvent.
3) Web Page processes a received message if the message's tab identifier is its own identifier and the message identifier does not match the one that was received in the past.

In this way, the per-resource server push can be realized by adding a mechanism to realize "exactly-once message delivery" and "pub/sub resources for each tab" to RFC 8030.

V. Conclusion

In this paper, we first described the demand for the per-resource server push and considered the requirements for realizing it. Next, we introduced the technologies that may be used to realize the per-resource server push, and we selected appropriate technologies to realize each requirement from them. As a result, we found that RFC 8030 supports the most requirements of among related technologies. Then, we have shown that by adding some extensions to RFC 8030, it is possible to realize a general-purpose per-resource server push that combines scalability, exactly-once, and pub/sub.

As a future prospect, we will implement a prototype and Software Development Kit (hereinafter referred to as SDK). This research is still in the theoretical stage and its practicality has not been verified. It is necessary to implement the mechanism proposed in this research and verify its practicality. Also, the proposed mechanism is complicated and difficult to use, so it will not directly used. We plan to provide an SDK that hides complexity and makes it easier to use.

REFERENCES

[1] M. Pohja, "Server push with instant messaging," in Proc. 2009 ACM Symp. on Applied Computing (SAC '09), 2009, pp. 653-658.

[2] M. Pohja, "Server push for web applications via instant messaging," J. Web Eng. vol. 9, no. 3, pp. 227-242, Sept. 2010.

[3] K. Shuang and K. Feng, "Research on Server Push Methods in Web Browser based Instant Messaging Applications." J. Softw., vol. 8, no. 10, 2013, pp. 2644-2651.

[4] de Souza Soares, E. F., Thiago, R. M., Azevedo, L. G., de Bayser, M., da Silva, V. T., and Cerqueira, R. F. D. G, "Evaluation of Server Push Technologies for Scalable Client-Server Communication," In 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), pp. 1-10, Mar. 2018.

[5] M. Thomson, E. Damaggio and B. Raymor, "Generic Event Delivery Using HTTP Push," IETF, https://datatracker.ietf.org/doc/html/rfc8030, Dec. 2016.

[6] P. Beverloo and M. Thomson, "Push API," W3C, https://www.w3.org/TR/push-api, June 2021.

[7] Automattic, "Socket.IO," https://github.com/socketio/socket.io, Aug. 2021.

[8] R. Rai, "Socket. IO Real-time Web Application Development," Packt Publishing Ltd, 2013.

[9] Jmattheis, "Gotify," https://gotify.net, Aug. 2021.

[10] OpenSaaS Studio, "Plasma," https://github.com/opensaasstudio/plasma, Aug. 2021.

[11] P. Th. Eugster, P. A. Felber, R. Guerraoui and A. M. Kermarrec, "The many faces of publish/subscribe," ACM Computing Surveys, vol. 35, Issue 2, pp. 114-131, June 2003.

[12] S. Tarkoma, "Publish/subscribe systems: design and principles.," John Wiley & Sons, 2012.

[13] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, "Known issues and best practices for the use of long polling and streaming in bidirectional http," https://datatracker.ietf.org/doc/html/rfc6202, Apr. 2011.

[14] I. Hickson, "Server-Sent Events," W3C, https://www.w3.org/TR/eventsource, Jan. 2021.

[15] A. Melnikov and I. Fette, "The WebSocket Protocol," IETF, https://datatracker.ietf.org/doc/html/rfc6455, Dec. 2011.

[16] V. Pimentel and B. G. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," in IEEE Internet Computing, vol. 16, no. 4, pp. 45-53, July-Aug. 2012.

[17] A. Lombardi, "WebSocket: lightweight client-server communications," O'Reilly Media, Inc., 2015.

[18] A. Russell, J. Song, J. Archibald and M. Kruisselbrink, "Service Workers," W3C, https://www.w3.org/TR/service-workers, Nov. 2019.